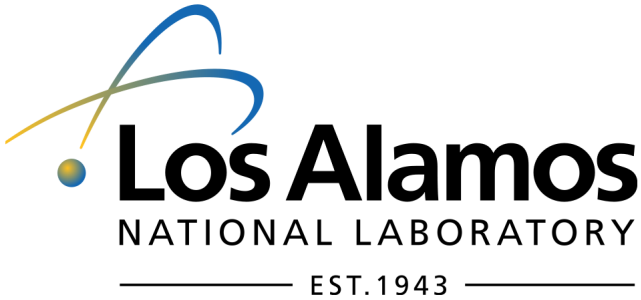
NOTE: Use these buttons to print or save the form. DO NOT use the browser tool bar. SAVE PRINT

LA-UR-11-05596

Approved for public release; distribution is unlimited.

Title:	IS&T 2011 Summer Co-Design School: Quasi Diffusion Accelerated Monte Carlo
Author(s):	Allen L. McPherson, Han Dong, Mahesh Ravishankar, Paul Sathre, Michael B. Sullivan, William Taitano, Jeffrey A. Willert, Timothy C. Germann, Dana A. Knoll, Bryan R. Lally, Patrick S. McCormick, Scott D. Pakin
Intended for:	Public release via web site (github.com/losalamos). Basic technical report that describes open source software LA-CC-11-076 (HILO). Not intended for conference or meeting publication.

Los Alamos NATIONAL LABORATORY EST.1943



IS&T 2011 Summer Co-Design School

Quasi Diffusion Accelerated Monte Carlo

Students:
Han Dong
Mahesh Ravishankar
Paul Sathre
Michael B. Sullivan
William Taitano
Jeffery Willert

Mentors:
Dr. Tim Germann
Dr. Dana Knoll
Dr. Bryan Lally
Dr. Pat McCormick
Dr. Allen McPherson
Dr. Scott Pakin

September 14, 2011

Contents

1 Boltzmann Transport Equation

The Boltzmann transport equation provides a continuum description of a 6D phase-space transport dynamics in time for a kinetic system. The equation is used frequently in the nuclear engineering and plasma physics community in order to study neutron, photon and plasma transport phenomenon in nuclear reactors and laboratory plasma settings. The generalized Boltzmann transport equation is shown in equation (1).

$$m\frac{\partial f}{\partial t} + m\vec{v}\cdot\nabla f + \vec{F}\cdot\nabla_v f = m\left(\frac{\partial f}{\partial t}\right)_c \tag{1}$$

m, f, t, $\vec{v}$ and $\vec{F}$ are mass of the particle, distribution function of the system, time, velocity and any external/internal force applied to the system, respectively [6]. The distribution function is a function of space, velocity and time, $f = f(\vec{r}, \vec{v}, t)$. The first term represents the change in the distribution function with respect to time. The second term represents the advection of distribution function in physical/configuration space, $\vec{r}$. The third term represents the advection of distribution function in velocity space. Finally, the fourth term represents a particle collision. This form is typically used in the plasma physics community. In the neutron transport community however, an alternative form is used in order to describe the transport of neutrons. The form of the Boltzmann transport equation used in the neutron and photon transport community, neglecting the acceleration term is shown in equation (2).

$$\frac{1}{v}\frac{\partial \psi}{\partial t} + \vec{\Omega}\cdot\nabla\psi = \left(\frac{1}{v}\frac{\partial \psi}{\partial t}\right)_c \tag{2}$$

v, ψ, t, Ω, F and E are the velocity magnitude, angular flux, time, solid angle, external/internal force applied to the system and energy respectively. The solid angle $\vec{\Omega}$ is a unit vector which represents the direction of the particle and is defined accordingly.

$$\vec{\Omega} = \sqrt{1-\mu^2}cos\phi\hat{i} + \sqrt{1-\mu^2}sin\phi\hat{j} + \mu\hat{k} \tag{3}$$

Where μ is referred to as the direction cosine, $\mu = \cos\theta$. The angular flux is a function of space, energy, solid angle and time, $\psi = \psi\left(\vec{r}, E, \vec{\Omega}, t\right)$. Each terms represent the identical physics of the original form of the transport equation. Additionally, since there are no long-range force of interest that acts on the particles in neutron and photon transport, the advection in velocity (or in this case, energy) space does not occur. This implies that the only mechanism in which the particle can lose energy is through collision.

$$\frac{1}{v}\frac{\partial \psi}{\partial t} + \vec{\Omega}\cdot\nabla\psi = \left(\frac{1}{v}\frac{\partial \psi}{\partial t}\right)_c \tag{4}$$

The condition of collision operator only allowing change in velocity of particle makes the modified form of the transport equation attractive in the discrete approach. The linearized, steady-state Boltzmann transport equation for neutron transport with collision is expressed accordingly.

$$\vec{\Omega}\cdot\nabla\psi + \Sigma_t\psi = \int_{\Omega=4\pi}\Sigma_s\left([\Omega',\Omega],[E',E]\right)\psi' d\vec{\Omega}' dE' + Q \tag{5}$$

Where the additional parameters, Σ_t, Σ_s and Q represents the total cross-section, scattering cross-section and a fixed internal source. E', Ω' and ψ' are the energy, solid angle and angular flux prior to collision. The first term represents the streaming of particles in physical/configuration space. The

second term represents the total collision interaction of neutrons. The third term represents the scattering collision interaction of neutrons. The final term represents the internal fixed source term that allows production of neutrons. With an assumption of isotropic scattering, isotropic source distribution and single energy, the transport equation takes the following simplified form.

$$\vec{\Omega} \cdot \nabla\psi + \Sigma_t\psi = \frac{1}{4\pi}\left(\Sigma_s\phi + Q\right) \tag{6}$$

Where ϕ is the scalar flux and is defined as the 0^{th} moment of the angular flux, ψ.

$$\phi = \int_{\Omega=4\pi} \psi d\vec{\Omega} \tag{7}$$

A final assumption of 1D in space provides the test problem in which our new algorithm is tested.

$$\mu\frac{\partial\psi}{\partial z} + \Sigma_t\psi = \frac{1}{4\pi}\left(\Sigma_s\phi + Q\right) \tag{8}$$

2 Deterministic Technique to the Solution of Transport Equation

There exists two general class of techniques in solving the original transport equation shown in equation (8). The two classes of techniques are the deterministic and stochastic approach. In this section, an overview will be provided in the traditional approach in solving the transport equation deterministically. The traditional approach in solving the transport equation deterministically is source iteration [4]. The concept of source iteration is formed behind converging the scattering source in the right hand side of equation (8).

$$\mu \frac{\partial \psi}{\partial z}^{k+1} + \Sigma_t \psi^{k+1} = \frac{1}{4\pi}\left(\Sigma_s \phi^k + Q\right) \tag{9}$$

$$\phi^k = \int_{\Omega=4\pi} \psi^k d\vec{\Omega} \tag{10}$$

Where k is the source iteration index. The concept is to solve for ψ^{k+1} in equation (9) from the known, initial guess scalar flux,ϕ^k then use the new ψ^{k+1} to calculate a new scalar flux, ϕ^{k+1}, from equation (10). This iteration is continued until a desired tolerance is met for either ψ or ϕ. The source iteration steps are provided below.

1. Solve for ψ^1 for the first iteration with initial guess of $\phi^0 = 0$ using equation (9).
2. Solve for ϕ^1 from equation (10).
3. Solve for ψ^{k+1} using ϕ^k using equation (9).
4. Solve for ϕ^k from equation (10).
5. Increment $k = k + 1$.
6. Check for convergence in ψ. If the system converged, end source iteration. If the system has not converged, repeat steps 3 to 5.

2.1 Discretization

In order to solve the transport equation numerically, an appropriate discretization (finite difference, volume, element) must be applied. For simplicity, a standard diamond differencing scheme can be applied in space and Gauss-quadrature integration scheme may be applied in angle.

$$\mu_n \frac{\psi^{k+1}_{i+1/2,n} - \psi^{k+1}_{i-1/2,n}}{\Delta z} + \Sigma_t \left(\psi^{k+1}_{i+1/2,n} + \psi^{k+1}_{i-1/2,n}\right) = \frac{1}{4\pi}\left(\Sigma_s \phi^k_i + Q\right) \tag{11}$$

$$\phi^k_i = \sum_{n=1}^{N} \omega_n \psi^k_{i,n} \tag{12}$$

Where the subscript i, n are the cell index and ordinate index and ω_n is the weight of quadrature point from Gauss quadrature. The weights are defined as the root of the Legendre polynomial of order n.

$$\omega_n = \frac{2}{\left(1 - \mu_n^2\right)\left[P'_n\left(\mu_n\right)\right]^2} \tag{13}$$

Now equation (11) can be solved through a standard transport sweep or a linear iterative solver such as GMRES [8].

The method is robust and will provide a converged solution eventually. However, for a certain class of problems, the method becomes very inefficient. The class of problem where the method

becomes inefficient is where the system size is large relative to the mean free path of neutrons and where the scattering interaction is dominant. The convergence becomes very slow and may require many source iteration until the solution actually converges. This is because the dominant physics of the problem is lagged between iteration. Additionally, for 3D problems, large amount of memory is required since the solution has to be stored for a 6D phase-space.

3 Stochastic Technique to the Solution of Transport Equation

An alternative to the deterministic source iteration is the stochastic Monte-Carlo approach, which does not explicitly require a descretization of the governing equation. For neutron transport problem, the transport equation is solved through a series of stochastic random number generation in order to determine the particle position, direction of flight, distance of travel and collision interaction. Based on a probability distribution function of the source from $Q(x)$ a random number can be generated to determine the location of particle. Based on the scattering cross section and total cross section, a probability of interaction can be determined. Finally, if the particle scatters, based on the scattering distribution function, a direction of scatter can be determined from yet another random number generation. Every time the particle collides, a physical quantity of interest is tallied. The process is enumerated accordingly.

1. Generate a random number and and determine the position of the particle based on the source probability distribution, $Q(x)$.

2. Generate a random number, R, to determine the direction of travel, μ.

$$\mu = -1 + 2R \tag{14}$$

3. Calculate the distance of travel from another random number generation.

$$x_f = x_0 - \mu \frac{ln|R|}{\Sigma_t} \tag{15}$$

4. Determine if the particle escaped the system. If so, end the particle history and restart from step 1. If the particle did not escape the system, generate a random number to check if the collision was a scattering or absorption.

   If $R \geq \frac{\Sigma_s}{\Sigma_t}$, absorption occurs. Perform tally then end particle history and restart from step 1.

   If $R < \frac{\Sigma_s}{\Sigma_t}$, scattering occurs. Perform tally then start from step 2.

5. Check if convergence is met. If not, return to step 1.

Note that all random numbers generated are assumed to be between 0 to 1.

3.1 Tally

This section will provide how the tallying of physical quantities are performed. There are two general approach to tallying: track length tally and surface crossing tally. The track length tally will calculate the total distance traveled by the particle within a cell while the surface crossing tally simply checks for the total number of particle that crossed a cell surface. Tally can be regarded as a moment calculation for a particle approach.

The 1D track length tally can be generalized accordingly.

$$\phi_{i,p}^n = \frac{1}{N_p \Delta V_i} \omega_p \Delta s_p \mu_p^{-1+n} \frac{|\mu_p|}{\mu_p} \tag{16}$$

Where n is the order of the moment defined as $n = 0, 1, 2, \cdots,$ N where N here is the largest moment of interest. Hence for scalar flux,ϕ, $n = 0$ and current, J, $n = 1$ etc. N_p, ΔV_i, ω_p, Δs_p and μ_p are the total weight of the system, volume of the i^{th} cell, weight of the particle, the total distance the

particle traveled within the i^{th} cell and the direction of the particle.

The surface crossing tally is simpler in calculation since no distance calculation within a cell is required. The 1D surface-crossing tally can be generalized accordingly.

$$\phi^n_{i,p} = \frac{1}{N_p \Delta A_i} \omega_p \mu_p^{-1+n} \frac{|\mu_p|}{\mu_p} \tag{17}$$

Where ΔA_i is the surface area of the ith cell surface that the particle crossed. The Monte-Carlo approach is regarded as a faithful simulation of the actual physics. The method is not subjected to any discretization errors and is able to reproduce a very accurate description of neutron behavior in the problem. However, similar to the deterministic source iteration approach, the method becomes very inefficient in large, scattering dominant problems. This is because the neutron would have to collide many times before it either escapes the system or gets absorbed. This makes the CPU cost per particle history be very expensive. In addition, for Monte-Carlo calculation, in order to smooth out the statistical noise and fluctuations, many many particle histories are required.

3.2 Sequential Implementation

The stochastic approach described above was implemented in C, to have a baseline comparison between the stochastic approach and the novel approach developed and discussed in the Section 4. The algorithm used is described below.

1. Divide the physical domain into `nc` cells.
2. Within each cell spawn particles such that the start position is uniformly distributed within the cell.
3. Calculate the direction and distance traveled by the particle using equations (14) and (15).
4. Calculate the end position of the particles and add its contribution to each cell it passes through.
5. If the particle doesn't stream out of the system then check if it is absorbed by generating a random number and checking if it is less that Σ_s.
6. If it is absorbed then goto Step 2, if it scatters, goto Step 3.

3.3 Multicore/multinode version

Since the particles in the stochastic process do not interact with one another, the approach is inherently data parallel. The Monte Carlo process also maps well to multicore CPUs since the threads don't execute in lock-step and the branch predictors in current hardware are sophisticated to reduce the overheads due to `if-else` constructs. MPI was used to parallelize the computation, with each process running on a logical core on the CPU. While using only MPI doesn't allow explicit use of shared memory space between the different threads on a core, modern MPI libraries are optimized to make use of the shared memory so as to reduce communication costs between the processes.

The approach used for an MPI implementation of Monte-Carlo is described below:

1. Each MPI process is allocated a contiguous set of cells to spawn particles in and track their evolution.
2. The contribution of all the particles simulated by a process are stored in arrays of length equal to the number of cells in the domain. In general, each process might have contributions to all the cells.
3. Reduce the contributions across MPI processes using `MPI_Reduce`.

Small Problem Size				
$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length = 10.0, Number of Cells = 100, Particles = 10000000				
No. of Processes	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	50765824.9	142296.99	582.37	
2	25336921.8	42428.66	1167.06	2.0004
4	15804749.9	53614.58	1870.93	3.2120
8	8719642.5	27664.05	3391.15	5.8220
16	4157151.1	35326.80	7112.96	12.2117
32	2193449.9	31119.74	13480.88	23.1442
48	1900629.5	283185.64	15557.81	26.7100

Table 1: Performance of the stochastic Method

4. Check if the values have reached convergence using the standard deviation as the convergence criteria. If it has not, return to Step 1.

The performance was evaluated on an 4 socket AMD Opteron processor with each socket supporting 12 cores (a total of 48 cores). Two problem sizes were tried. Only one-node was used for these experiments. Table 1 shows the time taken for the computation, along with the MFLOPS achieved and speedup by increasing the number of processes. The time taken for the computation was calculated by taking an average over 10 runs. The table also shows the standard deviation of the timings. It can be seem that as the number of processes increase, there is a huge variation in the execution time. This might be due to interference from other processes running on the system. As the number of threads used for the computation increases, these interferences have more impact on the run times.

3.4 OpenCL Implementation

The stochastic approach as described above, doesn't map well to GPGPUs. In such architectures, a group of 32 threads form a warp and maximum performance is achieved when all threads in a warp execute the same instruction on different data. A standard way of using GPGPUs for simulating Monte-Carlo is to have each thread simulate one particle. But since the particles might be absorbed or scattered based on random numbers, there can be a lot of thread divergence. Having an OpenCL implementation serves as a baseline to compare the benefits of algorithmic changes to the stochastic process. This section highlights the steps involved in such an implementation.

3.4.1 Number of Workgroups and Workitems

In OpenCL terminology workitems are the fundamental execution units on the GPU, also knows as threads. A workgroup is a group of workitems that can use shared memory to interact with each other and can synchronize with respect to each other. The configuration chosen for this problem is to set the number of workgroups and the number of workitems per workgroup equal to the number of cells in the domain. **NOTE:** To maximize performance , it is important to have as many workitems as possible. To do so, divide the domain into as many cells as necessary.

3.4.2 Tracing a particle

Each workgroup is tasked to spawn particles within one cell of the domain and trace their evolution. Each workitem starts off by spawning one particle and calculating its end position. Having done so all the threads synchronize and enter the tallying phase (described in 3.4.3). After exiting the tallying phase, each workitem checks if the particles is absorbed or goes out of the system. If it does, then that workitem, starts a new particles, otherwise it continues with the old particles. A counter in shared memory is used to keep track of the number of particles simulated by a workgroup.

Small Problem Size			
$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length = 10.0, Number of Cells = 100, Particles = 10000000			
	Time (μs)	**MFLOPS**	**SpeedUp**
Sequential	50165824.9	886.37	-
OpenCL - GTX 580	3709186	7970	13.69
OpenCL - Tesla 2050	5454904	5419.77	9.36

Table 2: Performance of the OpenCL kernel for stochastic methods

3.4.3 Tallying phase

After having streamed a particle for one flight, all the threads in the workgroup enter a tallying phase. The contribution made by particles simulated by a workgroup are stored in shared memory and is of size number of cells. To make tallying process efficient, multiple threads should not write to same location and the threads should write to consecutive locations in shared memory to reduce bank conflicts. The tallying process is divided into steps equal to the number of cells in the domain. In each step i, thread j adds its contribution to cell $(j+i)\%nc$ following which all threads synchronize before starting the next step. The synchronization is necessary since each thread has to check if it has a contribution to that cell. If it does, it calculates the value to be added, else a zero id added. After all the steps have been completed, the tallying process is complete and the threads go back to tracing the next/same particle.

3.4.4 Optimizing the tally phase

As mentioned above, during each step of the tally phase, each thread has to execute a bunch of conditionals to decide the contribution it needs to make. This can be expensive on the GPU. The overhead can be reduced by observing that there are 4 possible values that a thread might add. These are treated as four different states based on the target cell (the cell the thread adds its contribution to for the current step):

- State 0 : If the target cell is not crossed by the cell, then its adds a 0.
- State 1 : If the target cell is same as the starting cell of the particle, then it adds a value proportional to the distance traveled in that cell
- State 2 : If the target cell is greater than the starting cell, but less than the ending cell of the particle, then the value added is proportional to the cellsize since the particle travels the entire cell
- State 3 : If the target cell is same as the ending cell of the particle, then it adds a value proportional to the distance traveled in that cell

Further, the startcell and endcell of a particle can be manipulated such that the startcell is always lesser than the endcell. In such a situation, the threads can change states according the state diagram shown in Figure 1. The edge from State 1 to State 0 exist only if startcell is same as the endcell and the edge from State 1 to State 3 exist only if startcell is one less than the end cell. The shape of the state diagram per thread can be decided before entering the tallying phase. Therefore there is no need to check for this during the tally phase.

The thread divergence can be reduced since it is possible to calculate the number of steps a thread has to execute before it changes state. The new state can be decided cheaply based on the state diagram.

The performance of the OpenCL version was compared against the sequential implementation. Table 2 shows the speed up achieved by using the above driver on the GPU.

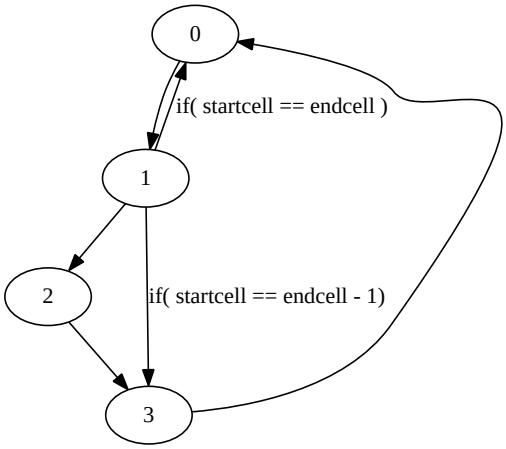


Figure 1: State Diagram to reduce thread divergence during tallying

4 Quasi-Diffusion-Acceleration

4.1 Background and History of Quasi-Diffusion-Acceleration

As discussed in section 2, the classic source iteration used to solve the transport equation deterministically suffers the slow convergence for cases where the system size is large relative to the neutron mean free path and when the collision interaction is scattering dominant. For many problems of interest in nuclear engineering (i.e. nuclear reactor), the system sizes are thousands of mean free path and the neutron suffers multiple collisions in the moderating medium (water, carbon, etc). In order to obtain a solution to the transport equation, the brute force deterministic source iteration approach is typically not used. Instead, an accelerator algorithm is used to obtain the solution using source iteration. The quasi-diffusion-acceleration (QDA) technique is typically used in combination with the source iteration to achieve the solution. The same algorithm will be extended to the stochastic Monte-Carlo approach to accelerate the solution to the underlying transport equation. The following section will discuss the background in the formulation of high/low order formulation of the governing equation and provide an overview as to how the lower-order moments solution is used to accelerate the higher-order transport equation.

4.2 Quasi-Diffusion-Acceleration Formulation

Consider the 1D, mono-energetic transport equation with a constant fixed homogeneous source, Q.

$$\mu\frac{\partial\psi}{\partial z}+\Sigma_t\psi=\frac{1}{4\pi}\left(\Sigma_s\phi+Q\right) \tag{18}$$

Now, a 0^{th} and 1^{st} moment is taken for the transport equation.

$$\frac{\partial J}{\partial x}+(\Sigma_t-\Sigma_s)\phi=Q \tag{19}$$

$$\frac{\partial E\phi}{\partial x}+\Sigma_t J=0 \tag{20}$$

Equation (19) and (20) are defined as the 0^{th} and 1^{st} moment of the transport equation, respectively. ϕ and J are defined as the 0^{th} and 1^{st} moment of the angular flux, ψ. E is defined as the ratio of the 2^{nd} and 0^{th} moment of the angular flux.

$$\phi=\int_{\Omega=4\pi}\psi d\vec{\Omega}=2\pi\int_{-1}^{1}\psi d\mu \tag{21}$$

$$J_x=J=\int_{\Omega=4\pi}\vec{\Omega}\psi d\vec{\Omega}=2\pi\int_{-1}^{1}\psi\mu d\mu \tag{22}$$

$$E_{xx}=E=\frac{\int_{\Omega=4\pi}\vec{\Omega}\vec{\Omega}\psi d\vec{\Omega}}{\int_{\Omega=4\pi}\psi d\vec{\Omega}}=\frac{2\pi\int_{-1}^{1}\psi\mu^2 d\mu}{2\pi\int_{-1}^{1}\psi d\mu} \tag{23}$$

Each terms are referred to as the scalar flux, current and the Eddington tensor. Note that in the 1D case, there's only a current in the x-direction and Eddington tensor is not exactly a tensor, but a factor. Now, equation (20) is solved for J.

$$J=-\frac{1}{\Sigma_t}\frac{\partial E\phi}{\partial x} \tag{24}$$

Inserting equation (24) into (21), the lower order equation for ϕ is constructed.

$$-\frac{\partial}{\partial x}\left(\frac{1}{\Sigma_t}\frac{\partial E\phi}{\partial x}\right)+\left(\Sigma_t-\Sigma_s\right)\phi=Q \tag{25}$$

Now a high/low order notation, HO and LO is used for both the original transport equation and the low order equation.

$$\mu \frac{\partial \psi}{\partial x}^{HO} + \Sigma_t \psi^{HO} = \frac{1}{4\pi} \left(\Sigma_s \phi^{LO} + Q \right) \tag{26}$$

$$-\frac{\partial}{\partial x} \left(\frac{1}{\Sigma_t} \frac{\partial E^{HO} \phi^{LO}}{\partial x} \right) + \left(\Sigma_t - \Sigma_s \right) \phi^{LO} = Q \tag{27}$$

Note that the classic diffusion equation is retrived for the special case of $E = 1/3$ where D = $1/3\Sigma_t$.

$$-\frac{\partial}{\partial x} \left(D \frac{\partial \phi^{LO}}{\partial x} \right) + \left(\Sigma_t - \Sigma_s \right) \phi^{LO} = Q \tag{28}$$

Now the focus will be given to the structure that exists between the two equations. It is interesting to note that by satisfying either one of the equation, the other equation is automatically satisfied as well. If equation (26) is solved for ψ^{HO}, equation (27) is satisfied via the Eddington tensor closure. Conversely, by solving equation (27) for ψ^{HO}, equation (26) can be solved through the closure ϕ^{LO}. This is generally true in the continuum. In the discrete however, an iteration must still be performed to converge the two system (higher/lower order). The following general steps are taken for the QDA scheme.

1. Solve equation (27) using the diffusion approximation, $E = 1/3$ for ϕ^{LO}.
2. Solve equation (26) using the ϕ^{LO} from the initial diffusion approximation in step 1.
3. Calculate E^{HO} using ψ^{HO} calculated from step 2.
4. Calculate for ϕ^{LO} using E^{HO}.
5. Calculate for ψ^{HO} using ϕ^{LO}.
6. Check for convergence in ψ^{HO}. If converged, end iteration. If not, repeat steps 4 - 6.

The method was first developed by [5] in the 1960's and have been a corner stone for many of the modern deterministic S_n solvers.

5 QDA Monte-Carlo (QDAMC) Formulation

Now the deterministic QDA concept will be extended to the stochastic Monte-Carlo system in order to develop a hybrid stochastic-deterministic accelerator. Similar to the deterministic system, a higher/lower order system of equation is constructed. However, the higher-order transport equation is now solved using the stochastic Monte-Carlo method instead of the deterministic transport sweep. In the QDAMC approach, unlike the classic Monte-Carlo (CMC), the particle does not explicitly "*scatter*". Instead, the scattering physics is taken care of in the lower-order PDE system. This removes the control if, elseif statement from the particle and greatly reduces the CPU time per history. In essence, each particle history is treated as an effective pure absorbing problem. Anytime the particle collides, irregardless of the collision type, the particle history is terminated and tally of physical quantities are performed.

In QDAMC, instead of calculating for the angular flux then calculating the Eddington tensor, the moment quantities are directly calculated by tallying the direction cosine of each particle in the cell. The lower order scalar flux, ϕ^{LO}, will still be used as a source for the Monte-Carlo calclation, but will be used to provide a non-uniform weight to the particles depending on which cell they are born in.

5.1 Particle Weight Assignment

Unlike the CMC, which in the analogue sense uses a constant particle weight, QDAMC is intrinsically non-analogue. The particle weight is assigned based on the sum of cell lower-order scattering source, $\Sigma_t \phi^{LO}$, and fixed inernal source, Q, which we will call, $\bar{Q}$.

$$\omega_p = \frac{\left|\bar{Q}_i\right| NP_{ref} \Delta V_i}{NP_{cell,avg}} \tag{29}$$

$$\bar{Q}_i = \Sigma_s \phi_i^{LO} + Q_i \tag{30}$$

Where ω_p, NP_{ref}, ΔV_i, $\mathrm{NP}_{cell,avg}$ are the weight of particle in the cell where ϕ_i^{LO} is defined, reference number of particles per unit volume that is supplied by the user, cell volume size and average number of particles per cell for the simulation. The non-constant particle weight is used in order to represent the effective total source distribution given a flat particle probability distribution function.

5.2 Boundary Condition

In using either the QDA or QDAMC method, a boundary condition must be used for the lower-order system. For the very first initial condition calculation, a classic vacuum boundary condition is used. The vacuum boundary condition is defined as follows.

$$\frac{1}{\phi_b^{LO}} \frac{\partial \phi_b^{LO}}{\partial x} = -\frac{1}{D} \tag{31}$$

$$D = \frac{1}{3\Sigma_t} \tag{32}$$

For the subsequent iteration, the higher-order solution provides closure to the lower order system in two ways: 1.) Eddington tensor and 2.) the boundary condition. In this study, we've tested the QDAMC algorithm with the following boundary condition and closure definition.

$$J_b^{LO} = \frac{J_b^{HO}}{\phi_b^{HO}} \phi_b^{LO} \tag{33}$$

$$J_b^{LO} = -\frac{1}{\Sigma_t} \frac{\partial}{\partial x} \left(E^{HO} \phi^{LO} \right)_b \tag{34}$$

With two equations and two unknown, the boundary condition can be closed by using information from the higher-order solution.

5.3 Discretization of Lower-Order System

Discretization scheme used for the lower-order system is described. Consider the 0^{th} order moment of transport equation shown in equation (19). By defining the solution as cell averaged quanity at cell centers, the following discretized approximate form of (19) is obtained using a finite volume scheme.

$$\frac{J_{i+1/2}^{LO} - J_{i-1/2}^{LO}}{\Delta x} + \left(\Sigma_t - \Sigma_s\right) \phi_i^{LO} = Q_i \tag{35}$$

Now, we discretize the 1^{st} moment at cell face for $J_{i+1/2}^{LO}$ and $J_{i-1/2}^{LO}$.

$$J_{i+1/2}^{LO} = -\frac{1}{\Sigma_t} \frac{E_{i+1}^{HO} \phi_{i+1}^{LO} - E_i^{HO} \phi_i^{LO}}{\Delta x} \tag{36}$$

$$J_{i-1/2}^{LO} = -\frac{1}{\Sigma_t} \frac{E_i^{HO} \phi_i^{LO} - E_{i-1}^{HO} \phi_{i-1}^{LO}}{\Delta x} \tag{37}$$

By substituting (36) and (37) into (35), we obtain the discretized form of the lower-order system for the internal nodes.

$$-\frac{1}{\Sigma_t}\frac{E^{HO}_{i+1}\phi^{LO}_{i+1} - 2E^{HO}_{i}\phi^{LO}_{i} + E^{HO}_{i-1}\phi^{LO}_{i-1}}{\Delta x^2} + (\Sigma_t - \Sigma_s)\,\phi^{LO}_i = Q_i \tag{38}$$

Now for the boundary condition, discretizing equations (33) and (34) at the left boundary.

$$J^{LO}_{1/2} = \frac{J^{HO}_{1/2}}{\phi^{HO}_{1/2}}\phi^{LO}_{1/2} \tag{39}$$

$$J^{LO}_{1/2} = -\frac{1}{\Sigma_t}\frac{E^{HO}_{1}\phi^{LO}_{1} - E^{HO}_{1/2}\phi^{LO}_{1/2}}{\Delta x/2} \tag{40}$$

Through few algebraic manipulation, $\phi^{LO}_{1/2}$ and hence $\mathrm{J}^{LO}_{1/2}$ is solved.

$$\phi^{LO}_{1/2} = -\frac{E^{HO}_{1}}{\Sigma_t \Delta x/2}\left(\frac{J^{HO}_{1/2}}{\phi^{HO}_{1/2}} - \frac{1}{\Sigma_t}\frac{E^{HO}_{1/2}}{\Delta x/2}\right)^{-1}\phi^{LO}_1 \tag{41}$$

$$J^{LO}_{1/2} = \frac{J^{HO}_{1/2}}{\phi^{HO}_{1/2}}\left[-\frac{E^{HO}_{1}}{\Sigma_t \Delta x/2}\left(\frac{J^{HO}_{1/2}}{\phi^{HO}_{1/2}} - \frac{1}{\Sigma_t}\frac{E^{HO}_{1/2}}{\Delta x/2}\right)^{-1}\phi^{LO}_1\right] \tag{42}$$

Given the internal node discretization and boundary closures, a linear system of equations can be constructed such that the following linear system is solved for ϕ^{LO}.

$$\begin{gathered} A\phi^{LO} = Q \\ A \in \mathbb{R}^{N\times N} \\ \phi^{LO}, Q \in \mathbb{R}^{N\times 1} \end{gathered} \tag{43}$$

Where N is the number of cells, A is the coefficient matrix and Q is the internal fixed source defined at each cell center.

5.4 Treatment of the Higher-Order System

In this section, a detailed treatment of the higher-order system will be discussed. After the lower-order scalar flux, ϕ^{LO} is calculated at cell center, the higher-order system is initialized through the following process.

1. Determine the number of particles created per cell based on NP_{ref} and $NP_{cell,avg}$.
2. Place particle randomly within a cell.
3. Assign a particle weight using (29).
4. Assign a particle flight direction.

$$\mu_p = \text{-}1 + 2R$$

This initialization process effectively reflects the right-hand source of equation (18). Next, the particles are streamed, collided and tallied accordingly.

1. Calculate the distance traveled before collision accordingly.

$$x_{f,p} = x_{0,p} - \mu_p \frac{ln|R|}{\Sigma_t}$$

2. Determine which cells the particle crossed and tally ϕ^{HO}, J^{HO} and E^{HO} at boundaries using surface tallying (for boundary condition closure) and E^{HO} at cell centers using track-length tallying.

5.5 Summary of QDAMC

As expected, the QDAMC algorithm will resemble a familiar iteration procedure as the traditional QDA method used in the deterministic world. The iteration procedure is shown below.

1. Solve equation (27) using the diffusion approximation, $E = 1/3$ for ϕ^{LO}.
2. Place particle based on the total source distribution, $\bar{Q} = \Sigma_s \phi^{LO} + Q$.
3. Assign a direction of flight to particle.
4. Stream and collide particle.
5. Determine which cells the particle crossed and perform tallying for ϕ^{HO}, J^{HO} and E^{HO} on boundaries and cell centers.
6. Calculate for ϕ^{LO} using E^{HO}.
7. Repeat steps 2 to 5.
8. Check for convergence. If converged, end iteration. If not, repeat steps 2 -7.

The only difference between the traditional deterministic QDA and QDAMC is in the treatment of the higher-order system. In QDA, the higher-order system is solved via a transport sweep or a linear solver based on the discretized transport equation. In QDAMC, the whole equation is effectively solved using Monte-Carlo.

6 MATLAB Results

This section will present results on the new QDAMC algorithm for the simple 1D, monoenergetic transport problem discussed in the section 4. In order to study the performance of QDAMC in the parameter space of various problems, three cases were studied and compared with against the CMC. The three cases studied are: The large, medium and small system size case. With each case, three separate sub-cases were studied with varying Σ_s. As $\lim \frac{\Sigma_s}{\Sigma_t} \rightarrow 1$, more scatteing dominant the problem becomes. By varying Σ_s, it is possible to invesetigate different problem regimes for each physical system size.

6.1 Large System Case

For the large system size case, the following problem parameters were chosen.

- $\mathrm{L}_x = 100$
- $\Sigma_t = 10$
- $\Sigma_s = 9.999, 9.99, 9.9$
- $N_x = 100$
- tolerance $\Delta x||\tilde{\phi}||_2 \leq 0.05$
- $NP_{cycle,QDAMC} = 5{\times}10^4$
- $NP_{cycle,CMC} = 1{\times}10^2$

Where L_x, Σ_t, Σ_s, N_x, $\Delta x||\tilde{\phi}||_2$, $NP_{cycle,QDAMC}$ and $NP_{cycle,CMC}$ are the system length, total cross-section, scattering cross-section, number of spatial cells, L^2 norm of the relative difference of scalar flux, number of particle per cycle for the QDAMC and CMC approach. The number of particles per cycle for the QDAMC and CMC was chosen to be different. The QDAMC method is sensitive to the number of particles per cycle and a high enough number of particles per cycle is required in order to suppress significant amount of noise feed-back between the higher-order and lower-order system (will be elaborated in later section). Additionally, the CMC method is independent of number of particles per cycle, but each particle history may undergo many scattering collisions and allows much more statistics accumulation per particle history relative to QDAMC. This implies that if a same number of particles per cycle is used for the two methods, a very coarse resolution information on CPU time for convergence will be available. As an example, even if the QDAMC technique required 20 cycle and the CMC technique required only 1 cycle, the raw CPU time required to *perform* that number of cycle can be in the order 5 seconds for QDAMC versus 3 hours for CMC. The relative difference is calculated from a reference deterministic solution with identical problem parameters. As will be discussed in more detail in the later section, there are no convergence theory for this hybrid deterministic-stochastic accelerator method. In order to characterize the performance of the method in various problem parameter space, a temporary convergence criteria of comparing the L^2 norm was adopted. A table organizing the results of the large case is shown below.

It can be seen that upon convergence, in order for the CMC solution to approach the same level of accuracy as the QDAMC, approximately 550 times more CPU time was required. Collisions refer to the average number of collisions particle undergoes before it is either absorbed or leaks out of the system. As seen, as the scattering cross-section increases, the number of collisions increases for the CMC. For QDAMC, since the collision logic is treated in the lower-order system, the average number of collision is 1 for all cases. Iteration is the required number of cycle for the method to converge below a specified L^2 norm tolerance. Finally, the total flights refers to the product of average collisions, number of particle used per cycle and the iteration. This quantity will provide effectively, the total number of streaming of particles required for convergence.

Test Results: $\Sigma_t = 10$, $L_x = 100$						
	$\Sigma_s = 9.9$		$\Sigma_s = 9.99$		$\Sigma_s = 9.999$	
	CMC	QDAMC	CMC	QDAMC	CMC	QDAMC
CPU Time	75.65	1.71	358.2	1.44	843	1.53
Collisions	98.7	1	973.6	1	8945.9	1
Iterations	396	11	196	10	49	11
Total Flights	3.908×10^6	0.55×10^6	19×10^6	0.5×10^6	43.8×10^6	0.55×10^6

Table 3: Large system size results for varying scatter cross-section for the CMC and QDAMC. Note: QDAMC used 5×10^4 particle. per iteration while CMC used 10^2 particles.

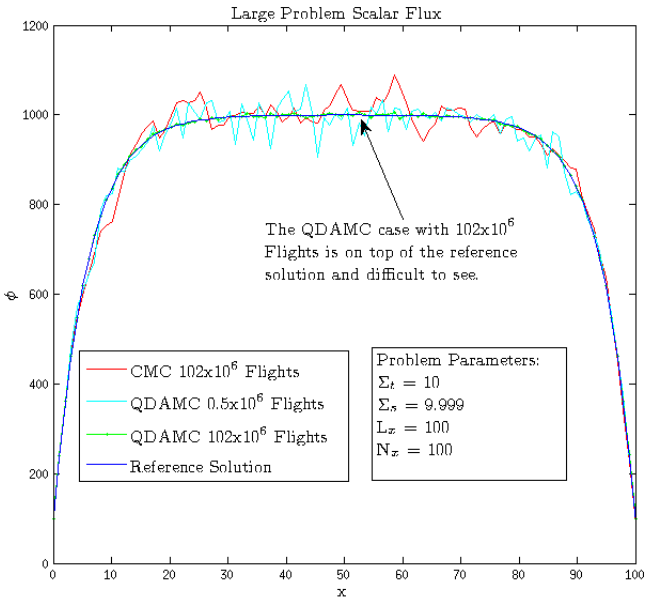


Figure 2: Large system scalar flux.

6.2 Medium System Case

For the medium system size case, the following problem parameters were chosen.

- $L_x = 10$
- $\Sigma_t = 10$
- $\Sigma_s = 9.999, 9.99, 9.9$
- $N_x = 100$
- tolerance $\Delta x||\tilde{\phi}||_2 \leq 0.005$
- $NP_{cycle,QDAMC} = 5\times10^4$
- $NP_{cycle,CMC} = 1\times10^2$

A table organizing the results of the medium case is shown below.

6.3 Small System Case

For the small system size case, the following problem parameters were chosen.

Test Results: $\Sigma_t = 10$, $L_x = 10$						
	$\Sigma_s = 9.9$		$\Sigma_s = 9.99$		$\Sigma_s = 9.999$	
	CMC	QDAMC	CMC	QDAMC	CMC	QDAMC
CPU Time	58.1	3.11	176.4	3.32	295.6	3.32
Collisions	89.0	1	649.2	1	1974.2	1
Iterations	256	17	106	18	56	18
Total Flights	2.28×10^6	0.85×10^6	6.89×10^6	0.9×10^6	11.05×10^6	0.9×10^6

Table 4: Medium system size results for varying scatter cross-section for the CMC and QDAMC. Note: QDAMC used 5×10^4 particle. per iteration while CMC used 10^2 particles.

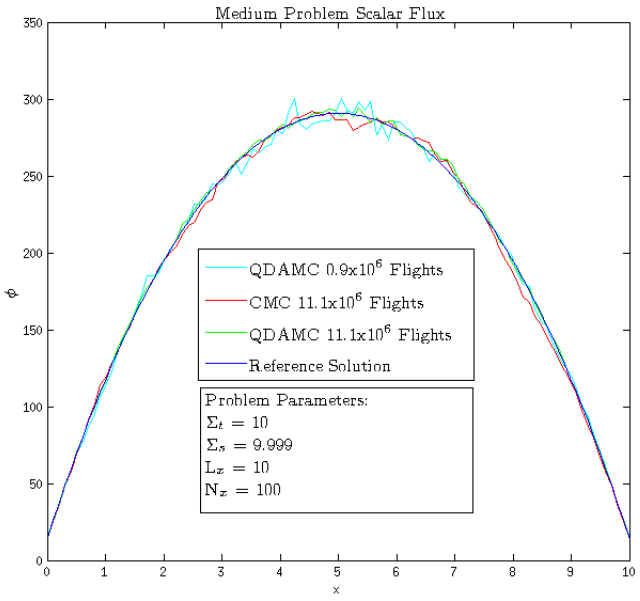


Figure 3: Medium system scalar flux.

- $L_x = 1$
- $\Sigma_t = 10$
- $\Sigma_s = 9.99, 9.9$
- $N_x = 100$
- tolerance $\Delta x||\tilde{\phi}||_2 \leq 0.001$
- $NP_{cycle,QDAMC} = 5\times10^4$
- $NP_{cycle,CMC} = 1\times10^2$

Only two sub-cases with Σ_s were tested since for the CMC, any higher scattering cross-section is meaningless as the system-size is now too small to allow anymore collisions per particle with increased scattering cross-section. A table organizing the results of the small case is shown below.

Test Results: $\Sigma_t = 10$, $L_x = 1$				
	$\Sigma_s = 9.9$		$\Sigma_s = 9.99$	
	CMC	QDAMC	CMC	QDAMC
CPU Time	23.453	10.08	25.85	19.10
Collisions	7.94	1	25.98	1
Iterations	950	23	332	43
Total Flights	0.71×10^6	1.15×10^6	0.86×10^6	2.15×10^6

Table 5: Small system size results for varying scatter cross-section for the CMC and QDAMC. Note: QDAMC used 5×10^4 particle. per iteration while CMC used 10^2 particles.

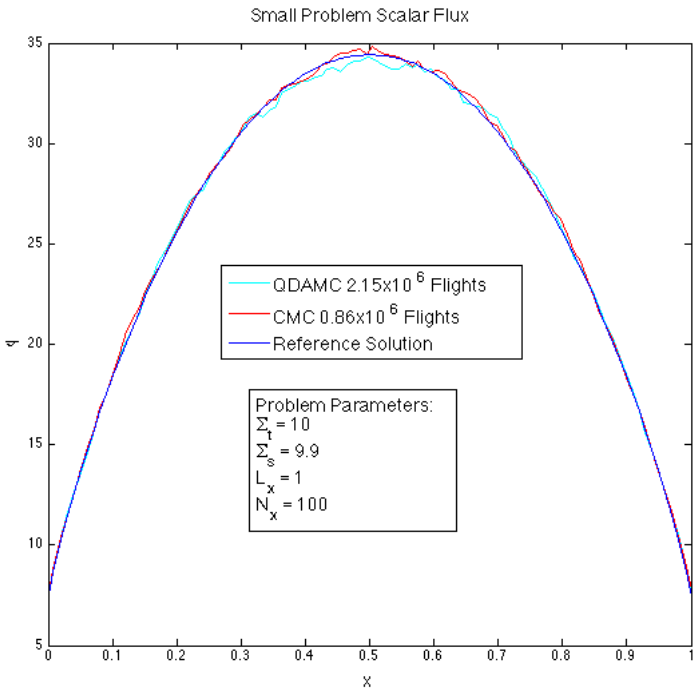


Figure 4: Small system scalar flux.

7 QDA-MC for MultiCore Architecture

While the QDA-MC approach was developed with the idea of handling the scattering of particles in the Lower Order equation, making the solution of Higher Order more amenable to GPUs; it is also true that the QDA-MC algorithm maps well to multi-core/multi-node architectures too. To evaluate the performance, a hybrid MPI/OpenMP approach was developed. This section discusses the approach used for parallelizing the code.

From the sequential code, it was apparent that, the Monte Carlo step to solve the higher-order equation was the most expensive part of the computation. Fortunately, the Higher Order Monte Carlo step is also extremely data parallel. The sequential implementation spawned particles uniformly across all the cells in the domain and tracks the cells through which they travel. Distributing the cells across MPI processes or OpenMP threads results in equal distribution of work in the HO system. The Lower Order system for the 1D case is extremely cheap and is done by one thread.

7.1 Using MPI

Using MPI for parallelizing the Higher Order solver involves the following steps

1. Solve the Lower order equation with the diffusion approximation on process 0.
2. Process 0, calculates $\bar{Q}$ in each cell based on results from the lower order solution, and broadcasts the values to the different processes using `MPI_Bcast`.
3. Each process spawns particles in the cells assigned to it with the weight of the particles determined by the value of $\bar{Q}$ in that cells.
4. Each process stores the contribution to ϕ, E at each cell, and to ϕ, E and J at the two boundaries.
5. The contributions from all the processes are combined using `MPI_Reduce`, with the result being put on process 0.
6. Process 0 then solves the lower order system using the values from the reduction for closure and checks for convergence.
7. If converged, process 0 sends a message to all processes to exit, else return to Step 2.

Table 6 shows the scaling of the QDA-MC code using MPI for two different problem sizes. For both the problems, while there is an almost initial linear speed up obtained, as the number of processes increase, it tapers of. This is due to the high variation in run times for MPI. By using all the 48 cores on the process a maximum performance of 9 GFLOPS for the small problem and 13 GFLOPS for the large problem is obtained. This is well below the peak performance of the machine. Even though modern MPI systems are sophisticated enough to use the shared memory if all processes are on the same core, there is a perofrmance hit observed due to communication costs.

7.2 Using OpenMP

While MPI allows use of a distributed memory system, for shared memory systems, the cost of communication can be cut down by using OpenMP and using the shared memory between the different threads. The steps involved in the OpenMP implementation is similar to MPI with a few minor modifications. The steps involved are as follows

1. Solve the Lower order equation with the diffusion approximation on thread 0.
2. Spawn OpenMP threads to start the QDA-MC solution.

Small Problem Size				
$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length = 10.0, Number of Cells = 100, Particles = 10000000				
No. of Processes	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	10675822.3	164769.97	631.72	
2	5528535.9	52333.43	1219.89	1.9310
4	2956620.3	95729.53	2281.04	3.6108
8	1700047.4	65268.60	3967.05	6.2800
16	1106856.8	194213.4	6093.09	9.64517
32	778089.4	94456.90	8667.62	13.7206
48	747197.4	178274.05	9025.97	14.2878
Large Problem Size				
$\Sigma_s = 9.999, \Sigma_t = 10.0$, System Length = 100.0, Number of Cells = 100, Particles = 40000000				
No. of Processes	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	33801229	116958.56	626.07	
2	17477552.2	882251.2	1210.82	1.9340
4	8765302.4	99823.87	2414.31	3.8562
8	4810222.6	211736.85	4399.42	7.0270
16	2697777.8	64257.35	7844.29	12.5293
32	1826685.6	219030.2	11610.43	18.5447
48	1617376.3	515903.08	13084.25	20.8988

Table 6: Performance of MPI version

3. Thread 0, calculates $\bar{Q}$ in each cell based on results from the lower order solution. $\bar{Q}$ resides in the shared memory and is accessed by all the threads.

4. Each thread spawns particles in the cells assigned to it with the weight of the particles determined by the value of $\bar{Q}$ in that cells.

5. Each thread stores the contribution to ϕ, E at each cell, and to ϕ, E and J at the two boundaries. These values are stored in arrays that are in shred memory space. To not have all threads writing to same memory locations, there are as many such arrays as there are threads, with each thread accessing only one such array.

6. The contributions from all the threads are combined using a tree reduction, with the arrays of thread 0 having the final reduced values.

7. Thread 0 then solves the lower order system using the values from the reduction for closure and checks for convergence.

8. If converged, then all threads exit the OpenMP parallel region and all spawned threads are destroyed. If not, then go to Step 3.

To get good performance while using OpenMP the following aspects have to be considered

- Each thread has to use its own random number generator object. Using a thread-safe random number generator would serialize the entire computation with threads waiting on each other to generate random numbers.

- The arrays used for each thread to store the contributions of the particles simulated by it needs to allocated such that they are on different cache lines. This would avoid false sharing of data between threads.

The Table 7 shows the run-times obtained for different number of threads. Comparing it with the MPI version, the OpenMP version is more scalable. This is due to elimination of communication constructs between the different threads. The tree reduction step while tallying is still a bottle-neck. For the small problem size a speed up of almost 24X is obtained, while for the large problem size, a speed up of almost 31X is obtained. Alos noteworthy is the standard deviation of the run-times. Due to elmination of communication systems between the threads, the variation in run times is much lesser when compared to the MPI version.

Small Problem Size				
$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length = 10.0, Number of Cells = 100, Particles = 10000000				
No. of Threads	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	10675822.3	164769.97	631.72	
2	5508699.1	165435.12	1224.28	1.9380
4	2995394.7	125112.9	2251.51	3.5641
8	1672604.5	223401.4	4032.14	6.3828
16	925145.2	42035.42	7289.86	11.5397
32	582612.2	68126.65	11575.2	18.3241
48	431958.2	41623.68	15613.03	24.7150
Large Problem Size				
$\Sigma_s = 9.999, \Sigma_t = 10.0$, System Length = 100.0, Number of Cells = 100, Particles = 40000000				
No. of Threads	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	33801229	116958.56	626.07	
2	17122685.4	267834.9	1235.91	1.9740
4	8856569	466659.93	2389.43	3.8165
8	4527149.4	147467.51	4674.5	7.4663
16	2473715.4	164247.9	8554.81	13.6641
32	1420319.4	96101.61	14899.58	23.7983
48	1059585.4	94526.57	19972.11	31.9004

Table 7: Performance of OpenMP version

7.3 Using Hybrid MPI-OpenMP

While MPI allows for implementation on distributed memory systems, allowing the code to run an huge multi-node computers, OpenMP uses shared memory to reduce communication overhead. The use of shared memory in OpenMP also reduces the total memory footprint of the computation, by using shared memory for read only data that is shared between the threads. From the previous sections , it was seen that a OpenMP approach gave better performance and scalability. Since modern super-computers contain many cores per node, an ideal approach would be use MPI processes across nodes, with each MPI process spawning OpenMP threads to exploit the multiple cores on each nodes. Even while using a single node with many cores, the memory space on the machine is partitioned between MPI processes. This might result in better cache locality since each process will allocate its data in a memory at a location that is "closer" to it. This is especially important with evolving NUMA architectures. Considering all these factors, a hybrid OpenMP-MPI version of the QDA-MC algorithmwas implemented as follows

1. Solve the Lower order equation with the diffusion approximation on thread 0 of process 0.
2. Spawn OpenMP threads on each MPI process to start the QDA-MC solution.
3. Thread 0 on process 0, calculates $\bar{Q}$ in each cell based on results from the lower order solution. $\bar{Q}$ resides in the shared memory of process 0. This array is broadcast to all the other MPI

processes which store the received array in their respective shared memory space.

4. Each thread on each process spawns particles in the cells assigned to it with the weight of the particles determined by the value of $\bar{Q}$ in that cells.

5. Each thread stores the contribution to ϕ, E at each cell, and to ϕ, E and J at the two boundaries. These values are stored in arrays that are in shared memory space of each MPI process. To not have all threads writing to same memory locations, there are as many such arrays as there are threads on each process, with each thread accessing only one such array.

6. The contributions from all the threads on an MPI process are combined using a tree reduction, with the arrays of thread 0 on all processes having the final reduced values.

7. The arrays belonging to thread 0 on each process are reduced using `MPI_Reduce`, with the final values residing in the arrays for thread 0 on process 0.

8. Thread 0 on process 0 then solves the lower order system using the values from the reduction for closure and checks for convergence.

9. If converged, then all threads on each process exit the OpenMP parallel region and all spawned threads are destroyed. If not, then go to Step 3.

Table 8 shows the run times achieved by sing 4 MPI processes and increasing the number of threads per process from 1 to 12. While the hybrid approach performs better than the pure MPI model, it is still slower than a pure OpenMP approach. But, it is an indication that while scaling to multiple nodes, a hybrid approach would give the best performance.

Small Problem Size				
$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length = 10.0, Number of Cells = 100, Particles = 10000000				
No. of Threads	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	2956620.3	95729.53	2281.04	3.6108
2	1850645.8	91896.85	3644.23	5.7687
4	1036984.2	182941.8	6503.65	10.295
8	641009.56	52209.54	10521.18	16.654
12	567316.89	66292.81	11887.85	18.8181
Large Problem Size				
$\Sigma_s = 9.999, \Sigma_t = 10.0$, System Length = 100.0, Number of Cells = 100, Particles = 40000000				
No. of Threads	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	8765302.4	99823.97	2414.31	3.8562
2	4710690	279831.4	4492.37	7.1754
4	2635279.2	116288.11	8030.33	12.8264
8	1659429.7	113606.19	12752.67	20.3691
12	1300413.3	198665.1	16273.41	25.9927

Table 8: Performance of Hybrid version

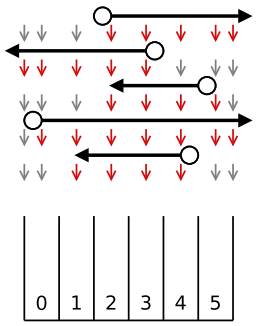


Figure 5: Schematic for Particlewise Tally

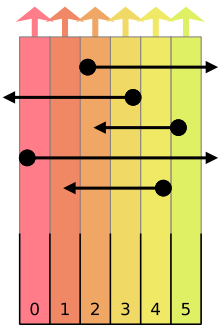


Figure 6: Schematic for Cellwise Tally

8 QDA-MC for GPUs

General Purpose Graphics Processing Unit (GPGPU) approaches to accelerating scientific applications represent a current major thrust in the field of High Performance Computing. For some applications it is possible to see speedups of several orders of magnitude over serial CPU execution; however, this performance often requires significant programmer effort to refactor algorithms to map more favorably onto GPU architectures. Further, such gains require highly data-parallel applications and are often limited by necessary control code and synchronization costs. A full discussion on the GPGPU paradigm is outside the scope of this document. For the current implementation we chose to use OpenCL. As a result, while we were targeting NVidia Fermi GPUs (GeForce GTX 580 and Tesla c2050), the implementation would be vendor agnostic, allowing for increased portability.

8.1 Particle Generation

The first step in the OpenCL implementation is to generate particles on the GPU. This requires use of a Pseudo-Random Number Generator (PRNG). An implementation of Martin Luscher's Ranlux PRNG implemented by Ivan Nikolaisen for AMD Cypress GPUs was chosen for this purpose since it allows per-thread non-correlated streams of PRNs. The process of generating particles is inherently parallel. In the Generation phase, each thread generated particles and stored it in global memory. The generation of particles entails calculating the value of the direction cosine μ, the start position of the particle and the end position of the particle.

8.2 Tallying

Once the particles have been generated, their contribution to different cells needs to be tallied. There were two approaches that were tried. In the first approach, which can be called as "particle-wise tally", each thread accumulated the contribution of a subset of particles into an array of size number of cells. To avoid serialization between threads, each thread was allocated an array of size equal to the number of cells in the domain. Since the amount of shared memory is not enough to hold all this data, this array was kept in global memory. The usage of global memory was reduced by generating the particles on the fly without explicitly storing their direction cosines and start/end positions. Having done that, the final answer is obtained by reducing the arrays across all threads. This is an inherently expensive process on the GPUs cause it requires traffic to global memory, without making use of the fast local memory available on the GPU. Further, the approach required expensive global synchronization constructs. The idea is illustrated by the Figure 5.

A complementary approach to the one described above was to, have each workgroup accumulate the contribution from all the particles to a specific cell in the domain. This approach, called "cell-wise tally" is illustrated in Figure 6. After generating all the particles and storing them in global memory during the generation phase, in the tally phase, each workgroup reads the information from all the particles generated and adds the contribution to one cell of the domain. The data from the

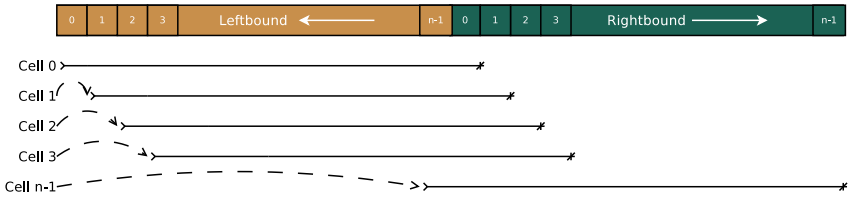


Figure 7: Data Layout of particle for improved tallying

$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length = 10.0, Number of Cells = 100, Particles = 10000000			
	Time (μs)	**MFLOPS**	**SpeedUp**
Sequential	10547426.2	886.37	-
OpenCL - GTX 580	565114.3	11934.19	13.46
OpenCL - Tesla 2050	900534.8	4506.46	7.489

Table 9: Performance of the OpenCL kernel for stochastic methods

generation phase could be laid out such that there is coalesced access to global memory. Further improvement was achieved by the use of "cl-fast-relaxed-math" optimization flag which allows the OpenCL kernel to use fused multiply add (FMA) operations.

It is apparent that with the "cellwise tally" approach, the GPU does a lot more work than the corresponding CPU implementation. Only a small subset of particles have contributions to any given cell, but since the direction of flight and distance traveled is stochastic in nature, this information is not available before actually scanning the particle. However, based on the problem at hand, we know that half of the particles in the domain travel left and half of them travel right. The right-bound particles "born" to the right of any given cell cannot possibly interact with said cell, and the same for left-bound particles born to the left, each cell could ignore just short of half of all particles that were guaranteed to be moving away from the cell. A modification to the generation phase was made such that all the left bound particles were stored contiguously followed by all right bound particles, as shown in Figure 7. Threads of a workgroup, say i, could scan the left-bound particles spawned in cells i to $n-1$ and right bound particles spawned in cells 0 to i, and accumulate the contribution to cell i. This way the workgroup cuts down on the amount of wasted work done by it.

The Table 9 gives the experimental evaluation of the above scheme. For the two test problems. The sequential time is measearued on an AMD Opeteron 6168 processor. Two GPUs were used for evaluation, a GTX 580 and Tesla 2050.

9 QDA-MC with CPU-GPU implementation

From the previous sections it is clear that while the GPU is extremely at generating particles, the tallying phase is a bottle-neck which makes the GPU less effective. On the other hand, the CPU is extremely fast at tallying the information from different particles. To make use of best of both worlds, a hybrid CPU GPU approach was explored. Two approaches were tried and are described below

9.1 MPI-OpenCL

In the initial version of the application, a simplistic and straightforward approach was taken. The application called a GPU kernel to generate all the particle data for that iteration, then the results are read back into a buffer on the CPU, the buffer is then scattered to all MPI tasks which do the rest of the calculation and tallying for that iteration. The main difference between this design and the pure MPI is that more communication overhead was added to the application, the overhead for GPU to CPU data transfer and the overhead for MPI_Scatter after the GPU finishes running the kernel.

In the second version, double buffers were added for asynchronous GPU to CPU transfers in order to hide the latencies such as waiting for GPU to finish its kernel and reading the data back from the GPU. An example of the double buffering is that on the GPU and CPU there is an array of (Number of Particles * 2). Initially, the particle generation kernel is called twice, once for the first section and once for the second section. Asynchronous read from GPU to CPU is called after each invocation of kernel call in order to overlap kernel execution with GPU to CPU data transfer. After this initialization, for each iteration in the while loop, we have an explicit wait event for the read of the 1th chunk of buffer to be read back into the CPU buffer completely. Next the data is scattered and each task then processes the data and tallies the result, we call the kernel and asynchronous read again for the 0th chunk since the data in the buffer is already used and can be safely overwritten. In the next iteration, we do the same except we're waiting and processing the 2nd chunk of the buffer. We then call the kernel and asynchronous read on for the 2nd chunk of the buffer since it can be safely overwritten. It can be observed during this process that data transfer and computation on the CPU and GPU can be overlapped by switching between the 1st and 2nd chunk of the big double buffered array. More details regarding this design is shown in Fig 8

9.2 OpenMP-OpenCL

In the initial version of the application, similar to MPI, a GPU kernel is called to generate all the particle data for that iteration, then the results are read back into a shared buffer on the CPU where each thread reads its own portion of the buffer to do the rest of the calculation and tallying until the next iteration. This implementation removes the scattering overhead. Similar to the MPI version , in the second version, double buffers and asynchronous GPU to CPU transfers are used in order to hide some latencies of waiting, the motivation and design is similar to the MPIGPU implementation except the fact that the scattering overhead is removed.

A Quad Core Intel Xeon CPU (3.2GHz) with an NVidia GTX 580 card was used to evaluate this design. Figure 9 shows the performance of the two version of OpenMP-OpenCL and the two version of MPI-OpenCL implementation as compared to the MPI and OpenMP implementation discussed before. It is clear that the hybrid CPU GPU implementation is the best performing on this machine. A future direction of work along this path is to evaluate the performance of this approach on the 48 core AMD Opteron Machine which in addition has an NVidia Tesla 2050 card attached to it.

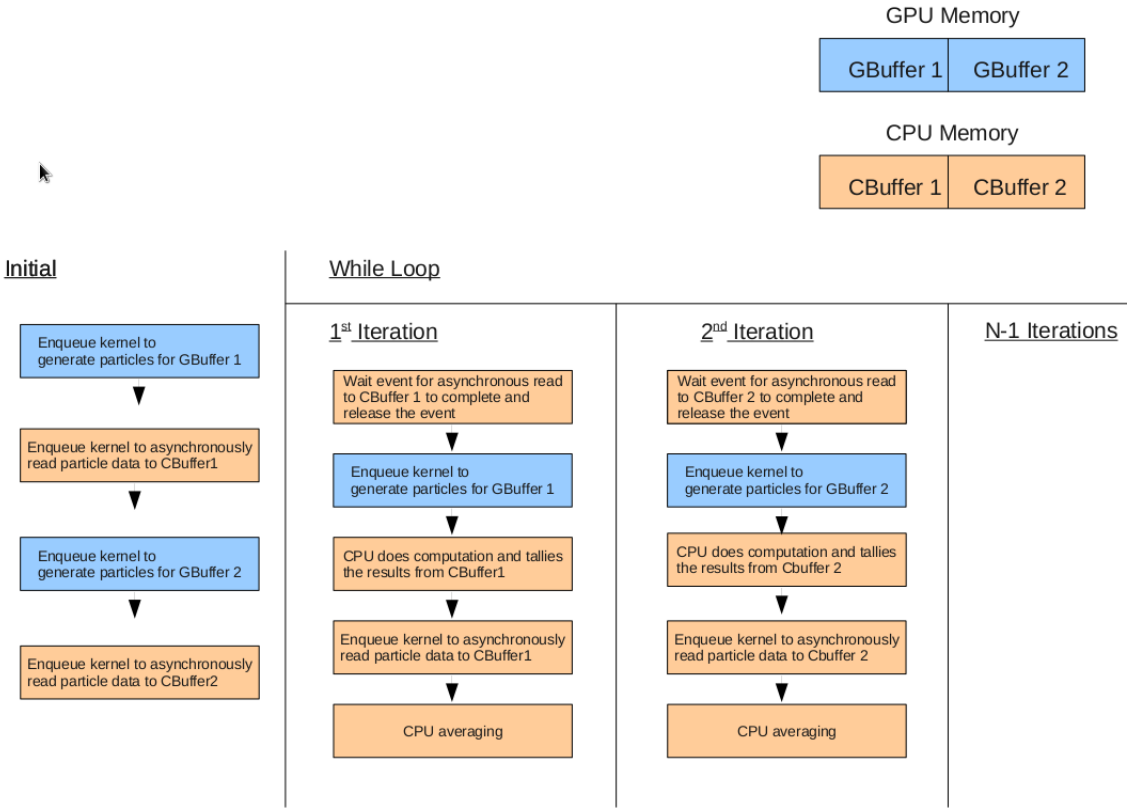


Figure 8: Schematic for double buffering in MPI-OpenCL implementation

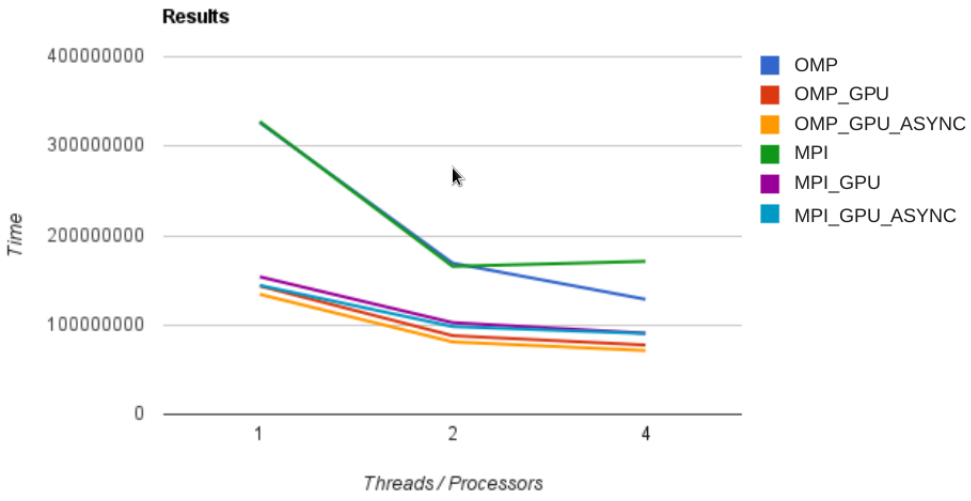


Figure 9: Comparison of Performance of the CPU-GPU hybrid approach with CPU approaches

10 Advantages and Disadvantages with QDAMC

During the testing of the method, it was discovered that the QDAMC algorithm was sensitive to both the mesh resolution as well as the number of particles used per cycle. Recalling equation (38), it is immediately seen that any noise from the higher-order system will be reflected down to the lower-order system through the Eddignton tensor. But in addition to the noise in the Eddington tensor, there is also the second derivative that in essence, *amplifies* the noise. This amplification is inverse proportional to the square of mesh size. By making the mesh size finer, the noise becomes stronger and when mesh is coarsen, the noise becomes weaker. It may seem like in order to deal with the amplification of noise, a coarser grid representation may be used. However, since now the higher-order Monte-Carlo source distribution is dependent on the lower-order solution, which is deterministic, there are two issues with this approach. First is the discretization error that is encountered in the lower-order equation. If a coarse grid is used in areas of solution where sharp gradients are encountered, such structures may not be captured using a coarse representation. Second is, with the current scheme, when the mesh size does not resolve the mean free path of neutron, not enough statistics gets accumulated using surface tallying schemes for boundaries. Hence for a statistically insignificant number of particles per cycle, the boundary quantities may not be tallied accurately. This introduces unphysical solution near boundaries, which feeds back into the higher-order Monte-Carlo calculation as the total source distribution, $\bar{Q}$.

Although there are issues and disadvantage with the QDAMC scheme, it is clearly seen from the results of three cases in tables 3, 4 and 5 that for large, scattering dominant system, the method wins over the CMC method. This is achieved by the acceleration of scattering source from the deterministic lower-order solution. In addition to the acceleration, the QDAMC method also accelerates the overall higher-order Monte-Carlo procedure through the removal of control logic from the collision interaction. The collision type is treated in the lower-order system. In addition, the method is independent of the scattering cross-section ratio. The iteration required to converge the solution below the specified convergence tolerance was roughly equal for the different sub-cases with varying scattering cross-section. However, it must be mentioned again that the method suffers from noise amplifications when using finer resolution mesh while the CMC method is independent of it. This implies that when using a high resolution mesh, the QDAMC method may not work as good as for the test cases ran in this study. A summary of advantages and disadvantages of the QDAMC method relative to CMC is given below.

- Advantages
  - Method is independent of scattering cross-section.
  - Removal of control logic from the collision.
  - Problem is independent of scattering cross-section.
- Disadvantages
  - Discretization error propogation from the lower-order solution.
  - Noise amplification between the higher-order and lower-order system.

11 Extension to 2D

In this section, a description of extension of QDAMC method to a 2D spatially non-homogeneous problem will be given.

11.0.1 2D Lower-Order System

Consider the original 3D transport equation with isotropic scattering and homogeneous internal fixed source, Q.

$$\vec{\Omega} \cdot \nabla \psi + \Sigma_t \psi = \frac{1}{4\pi} \left(\Sigma_s \phi + Q \right) \tag{44}$$

Where the directional unit vector, $\vec{\Omega}$ is defined as follows.

$$\vec{\Omega} = \sqrt{1-\mu^2} cos\phi \hat{x} + \sqrt{1-\mu^2} sin\phi \hat{y} + \mu \hat{z} \tag{45}$$

Now assuming a solution which is symmetrical in the xy-plane with no variation in the z-direction. Hence taking the 0^{th} and 1^{st} moment will yield the following set of lower-order equations.

$$\frac{\partial J_x^{LO}}{\partial x} + \frac{\partial J_y^{LO}}{\partial y} + \left(\Sigma_t - \Sigma_s \right) \phi^{LO} = Q \tag{46}$$

$$\frac{\partial \bar{E}^{HO} \phi^{LO}}{\partial x} + \frac{\partial \bar{E}^{HO} \phi^{LO}}{\partial y} + \Sigma_t \vec{J}^{LO} = 0 \tag{47}$$

Since $\vec{J}$ is obviously a vector, equation (47) can be expanded into the x and y components.

$$\frac{\partial \bar{E}^{HO} \phi^{LO}}{\partial x} + \Sigma_t J_x^{LO} = 0 \tag{48}$$

$$\frac{\partial \bar{E}^{HO} \phi^{LO}}{\partial y} + \Sigma_t J_y^{LO} = 0 \tag{49}$$

Where $\bar{E}^{HO}$ is now truly a tensor.

$$\bar{E} = \begin{bmatrix} E_{xx} & E_{xy} \\ E_{yx} & E_{yy} \end{bmatrix} \tag{50}$$

Hence equations (48) and (49) can be further expanded as follows.

$$\frac{\partial}{\partial x} E_{xx} + \frac{\partial}{\partial y} E_{yx} + \Sigma_t J_x^{LO} = 0 \tag{51}$$

$$\frac{\partial}{\partial x} E_{xy} + \frac{\partial}{\partial y} E_{yy} + \Sigma_t J_x^{LO} = 0 \tag{52}$$

Hence by solving for J_x^{LO} and J_y^{LO} and substituting the expression into equation (46), the following lower order system is derived.

$$\begin{aligned} &- \left[\frac{\partial}{\partial x} \frac{1}{\Sigma_t} \frac{\partial}{\partial x} E_{xx}^{HO} \phi^{LO} + \frac{\partial}{\partial x} \frac{1}{\Sigma_t} \frac{\partial}{\partial y} E_{xy}^{HO} \phi^{LO} \right. \\ &\left. + \frac{\partial}{\partial y} \frac{1}{\Sigma_t} \frac{\partial}{\partial x} E_{yx}^{HO} \phi^{LO} + \frac{\partial}{\partial y} \frac{1}{\Sigma_t} \frac{\partial}{\partial y} E_{yy}^{HO} \phi^{LO} \right] + \left(\Sigma_t - \Sigma_s \right) \phi^{LO} = Q \end{aligned} \tag{53}$$

Without going into details in the discretization, since all quantities expanded in equation (53) are scalar quantities, the classic central differencing may be applied for both the second spatial derivatives

in the x and y direction as well as for the cross terms . The discretized lower-order equation is shown below.

$$
\begin{aligned}
-\Bigg\{\Bigg[\frac{1}{\Sigma_{t,i+1/2,j}}\left(\frac{E^{HO}_{i+1,j}\phi^{LO}_{i+1,j}-E^{HO}_{i,j}\phi^{LO}_{i,j}}{\Delta x^2}\right)+ \\
\frac{1}{\Sigma_{t,i+1/2,j}}\left(\frac{E^{HO}_{i+1,j+1}\phi^{LO}_{i+1,j+1}+E^{HO}_{i,j+1}\phi^{LO}_{i,j+1}-E^{HO}_{i+1,j-1}\phi^{LO}_{i+1,j-1}-E^{HO}_{i,j-1}\phi^{LO}_{i,j-1}}{4\Delta x\Delta y}\right)\Bigg]- \\
\Bigg[\frac{1}{\Sigma_{t,i-1/2,j}}\left(\frac{E^{HO}_{i,j}\phi^{LO}_{i,j}-E^{HO}_{i-1,j}\phi^{LO}_{i-1,j}}{\Delta x^2}\right)+ \\
\frac{1}{\Sigma_{t,i-1/2,j}}\left(\frac{E^{HO}_{i,j+1}\phi^{LO}_{i,j+1}+E^{HO}_{i-1,j+1}\phi^{LO}_{i-1,j+1}-E^{HO}_{i,j-1}\phi^{LO}_{i,j-1}-E^{HO}_{i-1,j-1}\phi^{LO}_{i-1,j-1}}{4\Delta x\Delta y}\right)\Bigg]+ \\
\Bigg[\frac{1}{\Sigma_{t,i,j+1/2}}\left(\frac{E^{HO}_{i,j+1}\phi^{LO}_{i,j+1}-E^{HO}_{i,j}\phi^{LO}_{i,j}}{\Delta y^2}\right)+ \\
\frac{1}{\Sigma_{t,i,j+1/2}}\left(\frac{E^{HO}_{i+1,j+1}\phi^{LO}_{i+1,j+1}+E^{HO}_{i+1,j}\phi^{LO}_{i+1,j}-E^{HO}_{i-1,j+1}\phi^{LO}_{i-1,j+1}-E^{HO}_{i-1,j}\phi^{LO}_{i-1,j}}{4\Delta x\Delta y}\right)\Bigg]- \\
\Bigg[\frac{1}{\Sigma_{t,i,j-1/2}}\left(\frac{E^{HO}_{i,j}\phi^{LO}_{i,j}-E^{HO}_{i,j-1}\phi^{LO}_{i,j-1}}{\Delta y^2}\right)+ \\
\frac{1}{\Sigma_{t,i,j-1/2}}\left(\frac{E^{HO}_{i+1,j}\phi^{LO}_{i+1,j}+E^{HO}_{i+1,j-1}\phi^{LO}_{i+1,j-1}-E^{HO}_{i-1,j}\phi^{LO}_{i-1,j}-E^{HO}_{i-1,j-1}\phi^{LO}_{i-1,j-1}}{4\Delta x\Delta y}\right)\Bigg]\Bigg\}+ \\
\left(\Sigma_{t,i,j}-\Sigma_{s,i,j}\right)\phi^{LO}_{i,j}=Q_{i,j}
\end{aligned}
\tag{54}
$$

11.0.2 Boundary Condition

For the boundary condition, relative to the 1D implementation, a simpler implementation will be considered for the 2D case. In 1D, the boundary condition was implemented by solving the following two equations for J^{LO}_b and ϕ^{LO}_b.

$$
\begin{aligned}
J^{LO}_b &= \frac{J^{HO}_b}{\phi^{HO}_b}\phi^{LO}_b \\
-\frac{1}{\Sigma_t}\frac{\partial}{\partial x}E^{HO}_b\phi^{LO}_b
\end{aligned}
$$

However, for 2D, where the Eddington tensor is truely a tensor, the implementation becomes rather cumbersome. Instead, an alternative boundary condition implementation will be used. The new implementation will solely rely on the following closure.

$$
J^{LO}_b = \frac{J^{HO}_b}{\phi^{HO}_{b+1/2}}\phi^{LO}_{b+1/2} \tag{55}
$$

Notice that in this boundary condition implementation, the only unknown is in the lower-order scalar flux at the center of the boundary cell (i.e. b+1/2).

11.0.3 2D Higher-Order System

As we did in the 1D case, the higher-order system will also be treated using a stochastic Monte-Carlo techniques. Similar to the 1D case, assuming a uniform particle distribution, based on the total source defined as $\bar{Q}_{i,j}=\Sigma_s\phi^{LO}_{i,j}+Q_{i,j}$, the weight is assigned to particles born within a cell.

$$\omega_p = \frac{\left|\bar{Q}_{i,j}\right| NP_{ref}\Delta V_{i,j}}{NP_{cell,avg}} \tag{56}$$

Particle x and y position within a cell is chosen randomly and similarly for the direction cosine, μ_p and the azimuthal angle, ϕ_p.

$$\mu_p = -1 + 2R \tag{57}$$

$$\phi_p = 2\pi R \tag{58}$$

Where again, R is some random number ranging between 0 and 1. The subscript p implies that each variable are assigned for each particle. Now, the distance of flight is calculated identical to the 1D case.

$$\Delta s_p = s_{f,p} - s_{0,p} = -\frac{ln|R|}{\Sigma} \tag{59}$$

However, for a multi-material non-homogeneous case, the distance traveled is derived from the following attenuation probability.

$$P = exp\left[-\sum_{m=1}^{N} \Sigma_{t,m}\Delta s_m\right] \tag{60}$$

Where P, N, m, $\Sigma_{t,m}$ and Δs_m are the probability of particle making it to distance $\sum_{m=0}^{N} \Delta s_m$ without collision, total number of materials that the particle traverses, the material index, the m^{th} material total cross-section and the distance traved within the m^{th} material. Hence by using a random number R for P and solving for the N^{th} Δs_m, the distance traveled by particle in the last material is expressed as follows.

$$\Delta s_N = \frac{-ln|R| - \sum_{m=1}^{N-1} \Delta s_m}{\Sigma_{t,N}} \tag{61}$$

The tallying is different for the 2D case as the current is now a true vector and the Eddington tensor, $\bar{E}$, is a true 2 by 2 tensor. The track-length tallying for the scalar flux, current and Eddington tensor are shown below.

$$\phi_{i,j}^{HO} = \frac{1}{NP_{tot}\Delta V_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \Delta s_{p,i,j} \tag{62}$$

$$\vec{J}_{i,j}^{HO} = \frac{1}{NP_{tot}\Delta V_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \Delta s_{p,i,j} \vec{\Omega}_p \tag{63}$$

$$\bar{E}_{i,j}^{HO}\phi_{i,j}^{HO} = \frac{1}{NP_{tot}\Delta V_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \Delta s_{p,i,j} \vec{\Omega}_p \vec{\Omega}_p \tag{64}$$

Where NP_{tot}, $\Delta V_{i,j}$, $N_{p,i,j}$, ω_p and $\Delta s_{p,i,j}$ are the total number of particles in the system, the i,j cell volume, total number of particle that crossed into the i,j cell, the weight of the p^{th} particle and the total distance traveled by the p^{th} particle in the i,j cell. The current and Eddington tensor may be expanded accordingly.

$$\vec{J}_{i,j}^{HO} = \frac{1}{NP_{tot}\Delta V_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \Delta s_{p,i,j} \left(\sqrt{1-\mu_p^2} cos\phi_p \hat{x} + \sqrt{1-\mu_p^2} sin\phi_p \hat{y}\right) \tag{65}$$

$$\bar{E}^{HO}_{i,j}\phi^{HO}_{i,j} = \frac{1}{NP_{tot}\Delta V_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \Delta s_{p,i,j} \begin{bmatrix} \left(1-\mu_p^2\right) cos^2\phi_p & \left(1-\mu_p^2\right) cos\phi_p sin\phi_p \\ \left(1-\mu_p^2\right) sin\phi_p cos\phi_p & \left(1-\mu_p^2\right) sin^2\phi_p \end{bmatrix} \tag{66}$$

Similarly, for surface-tallying, the moment quantities can be calculated as follows.

$$\phi^{HO}_{i,j} = \frac{1}{NP_{tot}\Delta A_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \tag{67}$$

$$\vec{J}^{HO}_{i,j} = \frac{1}{NP_{tot}\Delta A_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \left(\sqrt{1-\mu_p^2} cos\phi_p \hat{x} + \sqrt{1-\mu_p^2} sin\phi_p \hat{y}\right) \tag{68}$$

$$\bar{E}^{HO}_{i,j}\phi^{HO}_{i,j} = \frac{1}{NP_{tot}\Delta A_{i,j}} \sum_{p=1}^{N_{p,i,j}} \omega_p \begin{bmatrix} \left(1-\mu_p^2\right) cos^2\phi_p & \left(1-\mu_p^2\right) cos\phi_p sin\phi_p \\ \left(1-\mu_p^2\right) sin\phi_p cos\phi_p & \left(1-\mu_p^2\right) sin^2\phi_p \end{bmatrix} \tag{69}$$

$\Delta A_{i,j}$ is the area of the surface of the i,j cell that the p^{th} particle crossed. Note that for a simple cartesian grid, there will be four surface for each cell. As easily seen, the Eddington tensor will collapse to a diagonal matrix in the diffusion limit where statistically, the particle contribution to the off-diagonal terms cancels out to yield zeros.

11.1 Preliminary MATLAB 2D Results

In this section, a preliminary 2D QDAMC results generated by a MATLAB code will be presented. Four cases are studied for the 2D case: 1. A simple homogeneous optically thick case, 2. a non-homogeneous optically thick case, 3. an optically thin homogeneous case and 4. a optically thin non-homogeneous case.

11.1.1 Homogeneous Optically Thick Case

For the homoeneous optically thick case, the system geometry is comprised of only a single material. The physical parameter of the homogeneous optically thick case is provided below.

- $L_x = 100$
- $L_y = 100$
- $N_x = 100$
- $N_y = 100$
- $\Sigma_t = 10$
- $\Sigma_s = 9.999$
- $Q = 1$

The scalar flux, current and Eddington tensor solution are shown in figure 11.1.1.

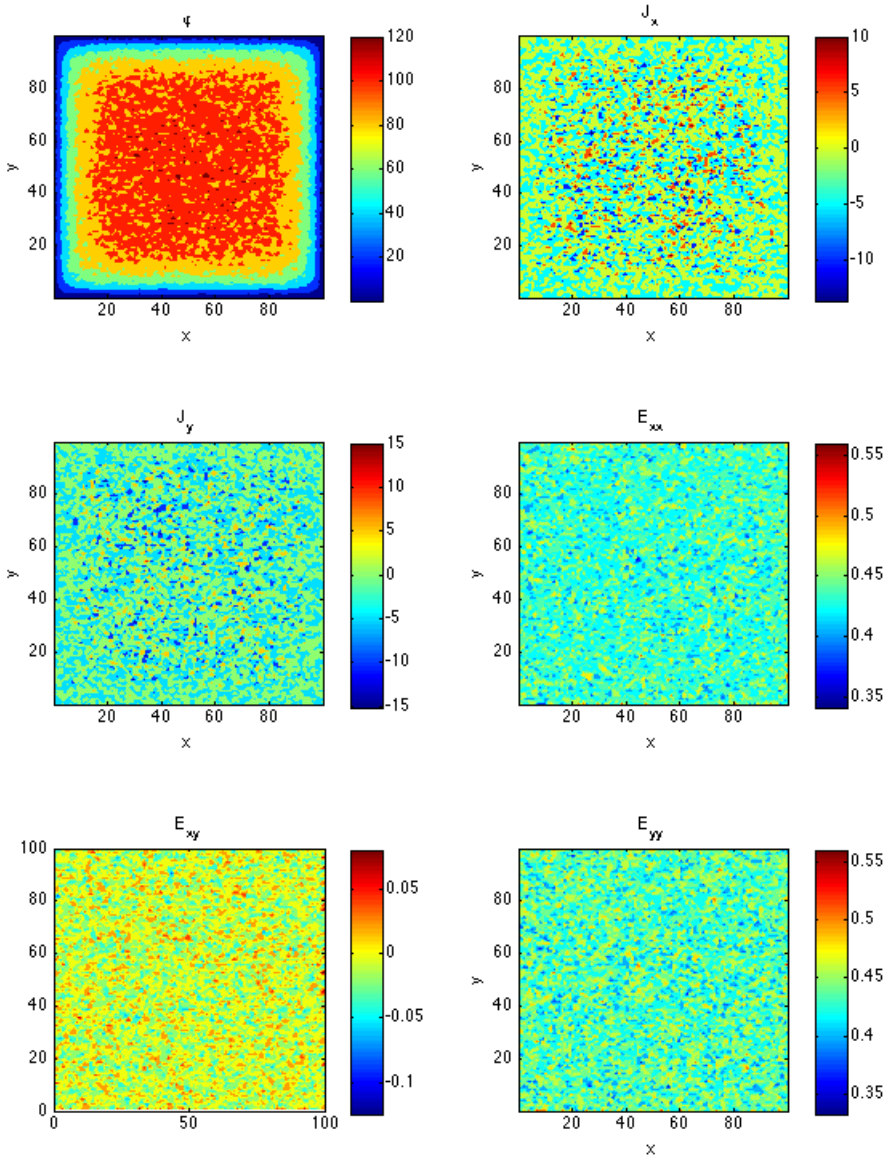


Figure 10: Solution for the 5th iteration. Homogeneous optically thick case.

11.1.2 Non-Homogeneous Optically Thick Case

For the non-homogeneous optically thick case, the system geometry is comprised of two material. The physical parameter of the homogeneous optically thick case is provided below.

- $L_x = 100$
- $L_y = 100$
- $N_x = 100$
- $N_y = 100$

- $\Sigma_{t,1} = 10$
- $\Sigma_{s,1} = 9.999$
- $\Sigma_{t,2} = 1$
- $\Sigma_{s,2} = 0.1$
- $Q_1 = 1$
- $Q_2 = 1$

The materials are defined in the following spatial regions.

$$\begin{array}{ll} material_2 & if \quad L_x/4 \leq x \leq L_x - L_x/4 \quad and \quad L_y/4 \leq y \leq L_y - L_y/4 \\ material_1 & if \qquad\qquad otherwise \end{array}$$

The scalar flux, current and Eddington tensor solution are shown in figure 11.1.2.

11.1.3 Homogeneous Optically Thin Case

The optically thin homogeneous case will have the following problem parameters.

- $L_x = 1$
- $L_y = 1$
- $N_x = 100$
- $N_y = 100$
- $\Sigma_t = 10$
- $\Sigma_s = 9.9$
- $Q = 1$

The scalar flux, current and Eddington tensor solution are shown in figure 11.1.3.

11.1.4 Non-Homogeneous Optically Thin Case

For the non-homogeneous optically thin case, similar to the thick case, the system is comprised of two materials. The problem physical parameters are given below.

- $L_x = 1$
- $L_y = 1$
- $N_x = 100$
- $N_y = 100$
- $\Sigma_{t,1} = 10$
- $\Sigma_{s,1} = 9.9$
- $\Sigma_{t,2} = 1$
- $\Sigma_{s,2} = 0.1$
- $Q_1 = 1$

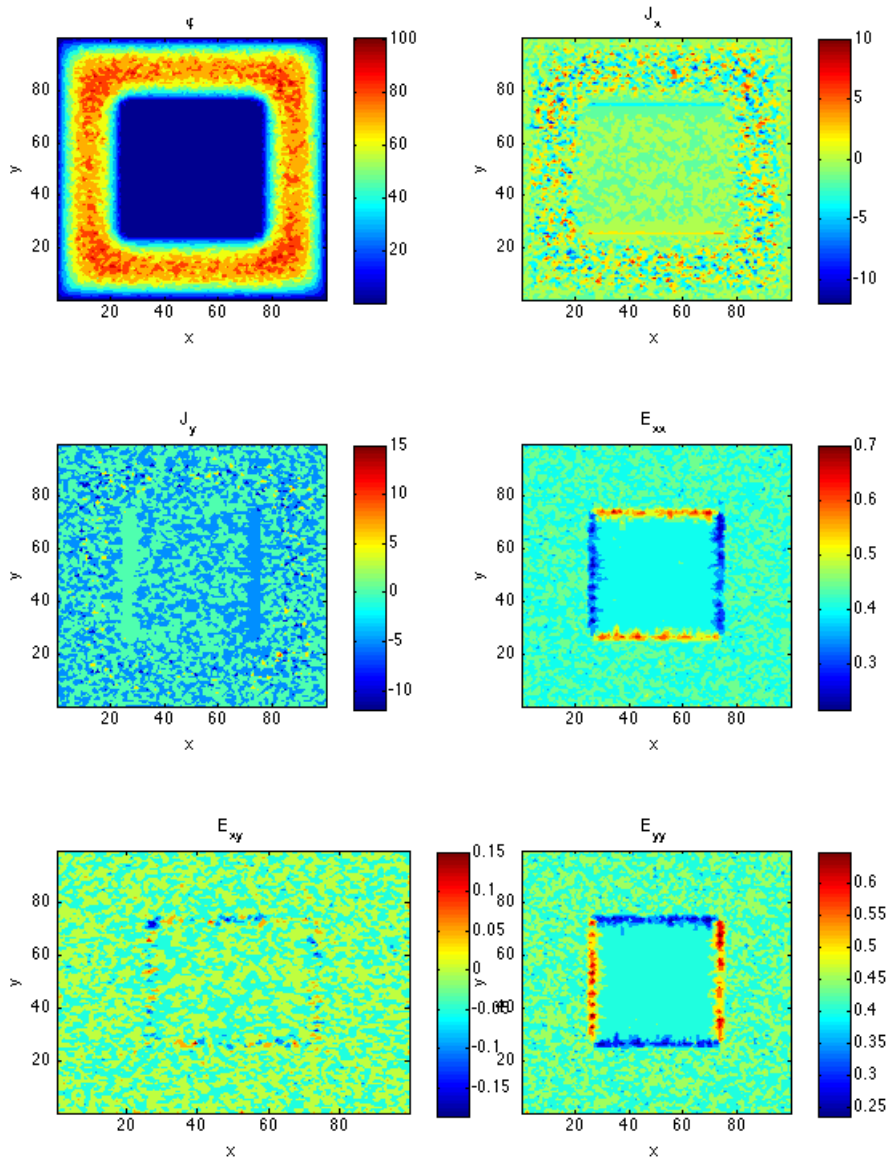


Figure 11: Solution for the 5th iteration. Homogeneous optically thick case.

- $Q_2 = 1$

The materials are defined in the following spatial regions.

$$\begin{array}{ll} material_2 & if \quad L_x/4 \leq x \leq L_x - L_x/4 \quad and \quad L_y/4 \leq y \leq L_y - L_y/4 \\ material_1 & if \qquad\qquad otherwise \end{array}$$

The scalar flux, current and Eddington tensor solution are shown in figure 11.1.4.

11.2 2D Serial implementation

The implementation of 2D QDAMC is similar to the implementation of 1D case. The steps involved are:

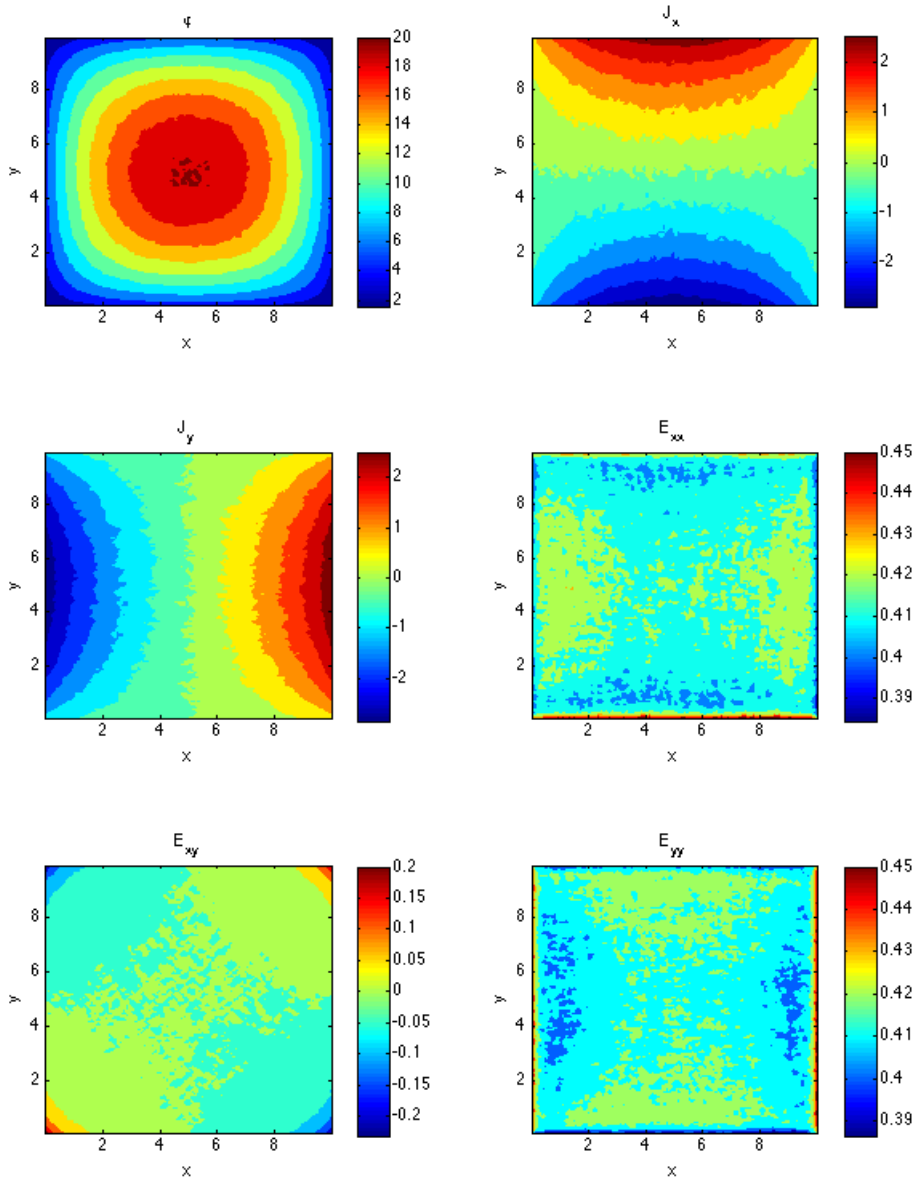


Figure 12: Solution for the 5th iteration. Homogeneous optically thin case.

1. Solve the Lower order Equation with a diffusion approximation
2. Use the solution of ϕ from the lower order system to calculate $\bar{Q}$.
3. Use Monte Carlo to solve the higher order equation with $\bar{Q}$ used to determine weights of the particles.
4. Compute the Eddington Tensor at all the cells, and values of Eddington tensor, ϕ and J at the boundaires.
5. Use the values from Step 4 to solve the Lower Order equation.

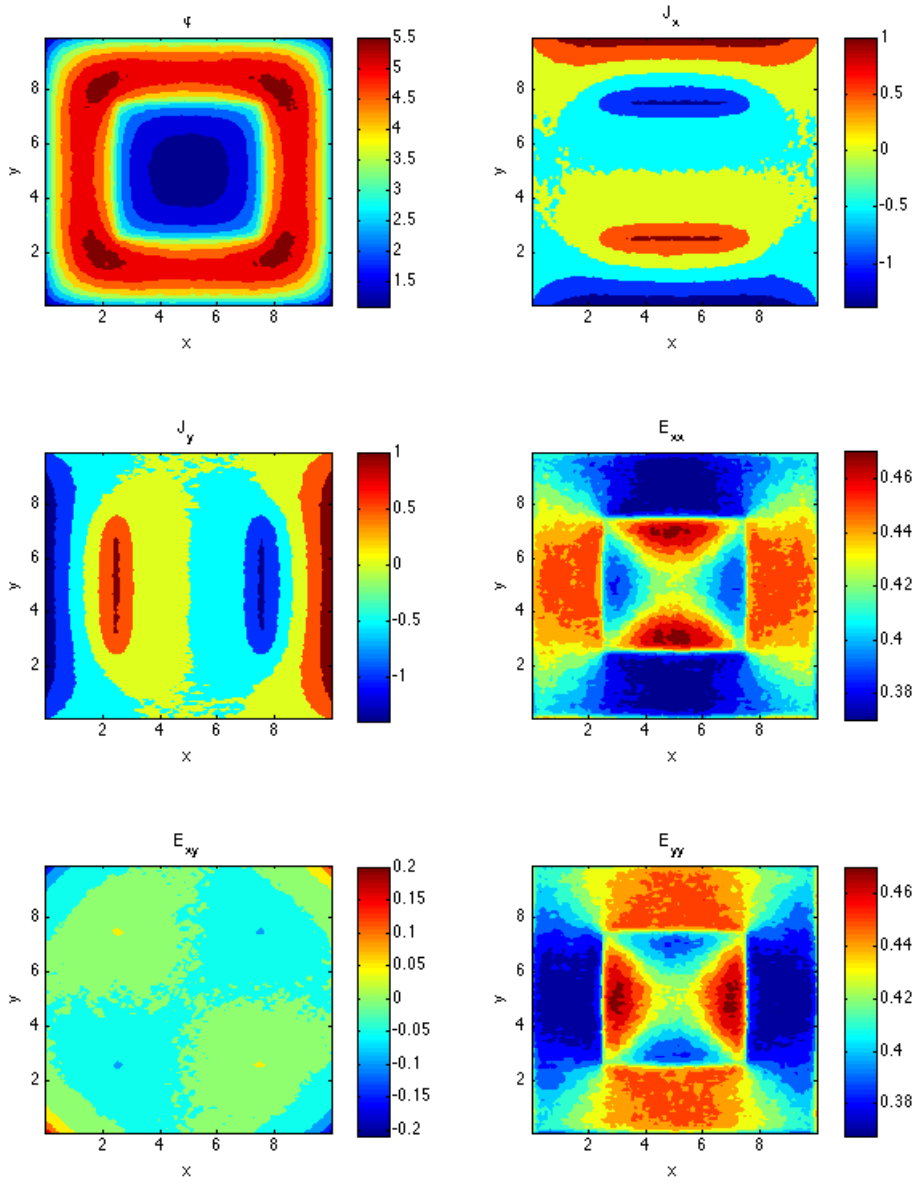


Figure 13: Solution fo the 5th iteration. Non-homogeneous optically thin case.

6. Check for convergence. If not converged, goto Step 2

The Lower Order equation is solved using the process described in Section 11.0.1.

The higher order system is solved by spawning an equal number particles in each cell, using a uniform distribution to spread out the start positions across the cell. Generate random numbers to calculate the direction cosine μ_p, the azimuthal angle, ϕ_p and the distance travelled, Δs_p using equations (57), (58) and (59). Using these values, the cell in which the particle ends its flight can be calculated. But unlike the 1D case where by knowing the start cell and the end cell it was possible to determine all the cells and the distance in each cell that a particle travels through, in 2D these need to be explicitly calculated. A fast method for calculating the cells through which a particle travels

and the distance travelled in each cell for both 2D and 3D rectangular grids is described in [1].

12 Future Work and Focus

During the development of the method, several advantages and disadvantages were discovered for the QDAMC method. The discretization error between the higher/lower order system may be addressed using a straight forward technique that enforces consistency between the two system, studied in detail in [3]. However, the issue of noise amplificaiton between the two system may be of focus for further future study. Finally, an accurate representation of the solution at boundaries are crucial in order to provide an accurate representation of lower-order source. Finally, a ground-work on a simple non-homogeneous 2D extension of the QDAMC algorithm is presented in this section.

12.1 Consistency Term

The consistency enforcing technique was developed to enforce consistency between the higher-order and lower-order system using non-linear diffusion acceleration of the deterministic transport equation. Recallig the original 1D transport equation and the accompanying lower-order equation.

$$\mu\frac{\partial\psi}{\partial x}+\Sigma_t\psi=\frac{1}{4\pi}\left(\Sigma_s\phi^{LO}+Q\right)$$

$$-\frac{1}{\Sigma_t}\frac{\partial^2}{\partial x^2}E^{HO}\phi^{LO}+\left(\Sigma_t-\Sigma_s\right)\phi^{LO}=Q$$

The two systems are equal in the continuum limit. However in the discrete limit, due to the difference in the discretization/numerical integration used between the two system, a small but finite inconsistency arise in the quantity for the scalar flux, ϕ. In order to enforce consistency between the two system, either we can force the higher-order system to follow the lower-order system, or vise-verse. The two approaches have their distinct advantage and disadvantage. The approach of enforcing the moment of the higher-order system to follow the lower-order solution may be used for variance reduction purposes [7]. In contrast, enforcing the lower-order system to follow the higher-order system can be used for enforcing consistency between the two system and simultaneously achieve an acceleration of the higher-order system [3]. In achieving acceleration of the higher-order Monte-Carlo solution, it is necessary to enforce the lower-order solution to follow the higher-order solution. First consider the modified higher/lower order system .

$$\mu\frac{\partial\psi^{HO}}{\partial x}+\Sigma_t\psi^{HO}=\frac{1}{4\pi}\left(\Sigma_s\phi^{LO}+Q\right) \tag{70}$$

$$\frac{\partial}{\partial x}\left[-\frac{1}{\Sigma_t}\frac{\partial}{\partial x}E^{HO}\phi^{LO}+\hat{D}\phi^{LO}\right]+\left(\Sigma_t-\Sigma_s\right)\phi^{LO}=Q \tag{71}$$

The $\hat{D}$ in the lower-order equation will be referred to as the consistency term which enforces the lower-order scalar flux to be equal to the higher-order scalar-flux down to the specified convergence tolerance. Note that the first term in the bracket is simply the expression for the current, J^{LO}. Hence the consistency term can be regarded as a correction to the current that enforces the consistency between the higher-order and lower-order solution. The expression for $\hat{D}$ is given as follows.

$$\hat{D}=\frac{\frac{1}{\Sigma_t}\frac{\partial}{\partial x}E^{HO}\phi^{HO}+J^{HO}}{\phi^{HO}} \tag{72}$$

With the current defined on cell face, the discretization of $\hat{D}$ is performed.

$$\hat{D}_{i+1/2}=\frac{\frac{1}{\Sigma_t}\frac{\partial}{\partial x}\left(E^{HO}_{i+1}\phi^{HO}_{i+1}-E^{HO}_{i}\phi^{HO}_{i}\right)+J^{HO}_{i+1/2}}{\phi^{HO}_{i+1/2}} \tag{73}$$

Where the higher-order terms are defined at their respective index. Taking the higher-order current, $J^{HO}_{i+1/2}$ as an example, the term will be defines as follows.

$$J^{HO}_{i+1/2} = 2\pi \int_{-1}^{1} \mu \psi^{HO}_{i+1/2} d\mu \tag{74}$$

Which is different from,

$$J^{HO}_{i+1/2} = \frac{2\pi \left(\int_{-1}^{1} \mu \psi^{HO}_{i} d\mu + \int_{-1}^{1} \mu \psi^{HO}_{i+1} d\mu \right)}{2} \tag{75}$$

However, ϕ^{HO}_i and E^{HO}_i are both defined at cell center using face averages.

$$\phi^{HO}_{i+1/2} = \frac{\phi^{HO}_i + \phi^{HO}_{i+1}}{2} \tag{76}$$

$$E^{HO}_{i+1/2} = \frac{E^{HO}_i + E^{HO}_{i+1}}{2} \tag{77}$$

$$\tag{78}$$

These definition of where the higher-order terms are defined are critical in enforcing consistncy between the two system. This is because the discretization of higher-order *solution* must follow *exactly* that of the lower-order system.

12.2 Filtering

While the consistency term provides consistency between the two system and addresses the discretization issue, the issue of noise amplification still remains with the QDAMC method which may make it unattractive for problems that require high spatial mesh-resolution. In this section, a suggestion will be provided which will attempt to suggest a general approach which may solve this problem. A promising technique in order to suppress the noise amplification is in the use of spatial filters. As an example, a simple binomial filter is presented. Consider a function f which is defined on a 1D spatial mesh which spans N_x. The binomial filter function will act on the function f in the following manner.

$$\tilde{f}_i = \frac{f_{i+1} + 2f_i + f_{i-1}}{4} \tag{79}$$

The filter will effectively diffuse the solution through a weighted averaging of neighboring cell value for f. This filter can also be considered a single pass of a Jacobi sweep. A multiple pass of a filter can be used to effectively damp out short wave-length noise. The use of spatial filters have been common in the plasma physics community using particle in cell approach [2]. There are issues with simple filters such as what is shown in (79) such as damping of sharp physical structures. However, those are issues that may be dealt with through more sophisticated filtering algorithms.

A 2D Classic Monte Carlo

A 2D Classic Monte Carlo implementation was done to serve as a baseline for comparison with the QDAMC solution. It also serves as a sanity check for the correctness of the higher order system in QDAMC, since the 2D Classic Monte Carlo implementation can be converted to simulate the Higher Order Scheme in QDA-MC by minor modifications. Following are the details of a sequential implementation and modifications to use multi-core/multi-node architecture.

A.1 Sequential implementation

The sequential 2D Monte Carlo implementation is very similar to the higher-order system used for QDA-MC described in Section 11.2. The main steps are outline here

1. Divide the 2D domain into a rectangular grid
2. Within each cell of the domain, spawn equal number of particles.
3. For a particle, use uniformly distributed random numbers to decide the start position within each cell.
4. Generate random number to calculate the values of direction cosine μ_p, the azimuthal angle, ϕ_p and the distance traveled, Δs_p using equations (57), (58) and (59).
5. Use the process of tracking a particle described in [1] to track the cells through which the particle travels and the distance traveled in each cell. Add contributions from that particle to the cell based on the distance traveled.
6. If the particle doesn't stream out of the domain, then check if it is scattered by generating a uniform random number between 0 and 1 and checking if it is less than Σ_s. If it scatters, using the end position as a new start position, return to Step 4. If not return to Step 3 to start a new particle.

A.2 MultiCore implementation

Just like the 1D Monte Carlo simulation, the 2D simulation is also inherently parallel. The same algorithm as described in Section 3.3 was used to parallelize the 2D code too. But note that such an approach would use data proportional to the number of cells per MPI process. This is an order of magnitude higher than that used for the 1D case. It is notable that for the 2D case, the average number of processes that have contribution to the same cell is lesser than the 1D case. Therefore, using a shared memory models OpenMP in conjunction with transactional memory models or atomic updates could result in reasonably good performance while significantly reducing the memory usage.

Table 10 shows the performance and scalability of the 2D Classic Monte Carlo code on a 4 socket, 12 cores per socket, AMD Opeteron Machine. The table shows the average execution time and the standard deviation across 10 runs. It can be seen that as the number of cores increase, the variation in time is also higher, due to interference from other system processes.

Small Problem Size				
$\Sigma_s = 0.99, \Sigma_t = 1.00$, System Length $= 10.0 \times 10.0$, Number of Cells $= 100 \times 100$, Particles $= 1000000$				
No. of Processes	**Mean Time (μs)**	**StDev (μs)**	**MFLOPS**	**SpeedUp**
1	60808478.1	109786.24	838.54	
2	30617262.8	104899.91	1665.41	1.9861
4	18634049.1	88510.49	2736.40	3.2633
8	9823110.3	95297.26	5190.85	6.1903
16	5158090.4	292708.66	9885.50	11.7890
32	2947911.0	430002.76	17291.10	20.6277
48	2314871.8	533402.56	22027.27	26.2686

Table 10: Scaling of 2D Classic Monte Carlo system

References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. Technical report, University of Toronto, Canada, 1987.

[2] C.K. Birdsall and A.B. Langdon. *Plasma Physics via Computer Simulation.* Taylor and Francis, 2005.

[3] K. Smith D.A. Knoll, H. Park. *Application of the Jacobian-Free Newton-Krylov Method to Nonlinear Acceleration of Transport Source Iteration in Slab Geometry.* Nuclear Science and Eng., 167, 122-132 (2011).

[4] J.J. Duderstadt. *Nuclear Reactor Analysis.* John Wiley and Sons, 1976.

[5] V. Ya. Gol'Din. *A Quasi-Diffusion Method for Solving the Kinetic Equation.* USSR Comput. Math. Math. Phys., 4, 136 (1967).

[6] S. Harris. *An Introduction ot the Theory of the Boltzmann Equation.* Dover Publications, Inc., 1999.

[7] G. Dimarco P. Degond and L. Pareschi. *The moment-guided Monte Carlo method.* Numerical Methods in Fluids., 67, 2, 189-213 (2011).

[8] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.